UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Introdução à Programação em Clusters de Alto Desempenho

por

Eduardo Henrique Rigoni
Rafael Bohrer Ávila
Marcos Ennes Barreto
Elgio Schlemer
César DeRose
Tiarajú Asmuz Diverio
Philippe O A Navaux

RP - 305 Outubro/1999

UFRGS - II - PPGC
Caixa Postal 15 064 - CEP 91501-970
Porto Alegre - RS - Brasil
Telefone: (051) 316 68 46
Fax: (051) 319 15 76
E-mail: diverio@inf.ufrgs.br

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API	Application Programming Interface
APL	Módulo do DPC++ responsável por funções básicas
BIP	Basic Interface for Programming
C++	Linguagem de Programação C++
DECK	Distributed Executive Communication Kernel
DOS	Disk Operating System
DPC++	Distributed Processing in C++
DSM	Distributed Shared Memory
GNU	Nome dado a licença de uso público
GPPD	Grupo de Processamento Paralelo e Distribuído
IP	Internet Protocol
LAM	Implementação do MPI feita pela universidade de Ohio.
LAN	Local Area Network
MCS	Implementação de MPI criado no Laboratório do MCS
MPI	Message Passing Interface
MPI CH	Implementação de MPI criado na Universidade de Massachusets
MPI FM	Message Passing Interface Fast Messages
MPP	Massively Parallel Processor
NFS	Network File System
NOW	Network Of Workstations
NUMA	Non Uniform Memory Access
PULC	Parastation User-Level Comunication
PVM	Parallel Virtual Machine
RPC	Remote Procedure Call
SMP	Symmetric MultiProcessor
TCP/IP	Transfer Control Protocol/Internet Protocol
UFRGS	Universidade Federal do Rio Grande do Sul
UDP	User Datagram Protocol
XDR	EXternal Data Representation

Resumo

Nos últimos anos tem-se investido na pesquisa de máquinas paralelas baseadas em clusters de multiprocessadores simétricos (SMP) por possuírem um custo relativamente mais baixo que as máquinas de arquiteturas maciçamente paralelas (MPP) além de serem mais flexíveis que essas. O objetivo desse trabalho é documentar e validar o uso das máquinas clusters, em especial, o cluster de alto desempenho da UFRGS e o ambiente de programação DPC++. A documentação inclui características do cluster da UFRGS, sendo dado uma ênfase especial às ferramentas disponíveis para programação de aplicações de alto desempenho, PVM, MPI e DPC++. A ferramenta DPC++ foi desenvolvida pelo Grupo de Processamento Paralelo e Distribuído e deriva do C++. Ela é orientada a objetos e de fácil adaptação por parte do programador. Também foram desenvolvidas aplicações visando a validação e a exemplificação do uso desse ambiente.

Palavras-Chave:

Processamento de Alto Desempenho, Processamento Paralelo, Processamento Distribuído, Cluster de Alto Desempenho, DPC++.

Abstract

In the last years one has invested in the research of parallel machines based on clusters of symmetrical multiprocessors (SMP) for possessing a cost relatively lower than the machines of architectures massive parallel (MPP) besides being more flexible than these. The objective of this work is to register and to validate the use of the machines clusters, in special, cluster of high performance of the UFRGS and the environment of programming DPC++. The documentation includes features of cluster of the UFRGS, being given a special emphasis to the available tools for programming of applications of high performance, PVM, MPI and DPC++. Tool DPC++ was developed by the Parallel Processing and Distributed group and drift of C++. It is objects oriented and of easy adaptation on the part of the programmer. Also the validation and the examplification of the use of this environment had been developed applications aiming at.

Keywords:

High Performance Computing, Parallel Processing, Distributed Processing, High Performance's Clusters, DPC++.

1 Introdução aos Clusters

Sistemas de processamento paralelo vêm se tornando cada vez mais populares em função da demanda por processamento de alto desempenho, exigido pelas diversas áreas da ciência (ex.: química, biologia, meteorologia). Infelizmente, os sistemas que oferecem a capacidade de processamento para satisfazer a essa demanda, representados pelas máquinas de arquiteturas maciçamente paralelas ou tem um custo elevado, ou são difíceis de programar, ou ambos. Em função disso, nos últimos anos, têm-se investido na pesquisa de máquinas paralelas baseadas em *clusters* de multiprocessadores simétricos por possuírem um custo relativamente mais baixo que as máquinas de arquitetura maciçamente paralelas além de serem mais flexíveis que essas.

Um cluster é uma máquina de alto desempenho que possui uma arquitetura baseada na reunião de um conjunto de estações de trabalhos independentes, interconectadas por uma rede de comunicação rápida, formando uma plataforma de execução de aplicações paralelas de alto desempenho.

A motivação pelo uso dessa arquitetura advém de diversos fatores, entre os quais o estado atual de desenvolvimento dos microprocessadores, permitindo a criação de processadores cada vez mais velozes com um custo relativamente baixo e a existência de redes de comunicação de dados de alto desempenho, comparáveis às redes proprietárias utilizadas em arquiteturas específicas. Outro fator é a disponibilidade de uma máquina desse tipo pelo grupo de Processamento Paralelo e Distribuído da UFRGS.

Atualmente, existem diferentes tipos de arquiteturas dedicadas à execução de aplicações paralelas, sendo que essas podem ser classificadas em três tipos:

- *Arquiteturas maciçamente paralelas (MPP)*: são arquiteturas que possuem processadores altamente poderosos e *links* de comunicação dedicados. Este tipo de arquitetura, chamada de supercomputadores ou arquiteturas dedicadas, apresentam um alto custo, devido aos recursos que oferecem. Como exemplo, pode-se citar o Intel Paragon e o IBM SP2.

- *Multiprocessadores simétricos (SMP):* são arquiteturas compostas por um conjunto de processadores iguais, que se comunicam, geralmente, através de uma mesma memória. O termo *simétrico* significa que todos os processadores são idênticos em termos de arquitetura interna e poder de processamento. Exemplos dessa arquitetura são os processadores Dual Pentium.

- *Redes de estações (NOW):* são arquiteturas que correspondem a um conjunto de estações de trabalho interligadas através de uma rede local (LAN) e que servem como plataforma de execução de aplicações distribuídas. Nesse tipo de arquitetura, a comunicação é feita por troca de mensagens entre as diversas aplicações que executam na rede. Esse tipo de arquitetura é largamente utilizado, tanto comercialmente como academicamente. Como exemplo, podemos citar Estações Sun interligadas por rede Ethernet.

Nesse contexto, um *cluster* pode ser caracterizado como uma plataforma alternativa, aliando o poder e a velocidade de processamento das arquiteturas dedicadas (MPPs) com a disponibilidade de recursos (hardware e software baratos) das redes de estações. É cada vez mais comum o uso de clusters compostos por multiprocessadores simétricos, como por exemplo, PCs com processadores Dual Pentium PRO ou Pentium II, como o caso do cluster existente na UFRGS.

Quando comparados com arquiteturas dedicadas, os *clusters* de multiprocessadores simétricos apresentam um grande número de vantagens. Eles são relativamente baratos (seus custos são menores que o custo de um supercomputador paralelo), eles oferecem uma boa relação custo/desempenho (porque todo o hardware e o software necessários estão à disposição), e, da mesma forma, suas volumosas vendas atraem investimentos diretos para o seu rápido melhoramento. Eles também permitem um desenvolvimento progressivo de aplicações, começando com apenas um processador, passando para multiprocessadores e, finalmente, usando um conjunto de estações de trabalho multiprocessadoras interconectadas por alguma rede de comunicação de dados local.

Pode-se caracterizar basicamente, duas classes de arquiteturas baseadas em clusters:

- *Arquiteturas homogêneas*: onde os nodos que compõem o cluster possuem a mesma arquitetura e sistema operacional, logo entendem as mesmas instruções sem a necessidade de conversão de dados a fim de possibilitar o processamento dos mesmos, em diferentes processadores. As arquiteturas homogêneas estão se tornando um padrão na área de clusters de alto desempenho, por serem mais simples de operar e por não apresentarem problemas ligados à conversão de dados entre diferentes sistemas operacionais e ou arquiteturas;

- *Arquiteturas heterogêneas*: onde os nodos que formam o cluster possuem processadores diferentes e, possivelmente, diferentes sistemas operacionais. Exigem a conversão de dados para que uma instrução possa processar em diferentes processadores. Apresentam problemas ligados à conversão de dados entre diferentes sistemas operacionais e ou arquiteturas

Além dessas classes de arquiteturas *cluster*, pode-se distinguir dois tipos de classificação quanto aos nodos que fazem parte do cluster:

- *Arquitetura simétrica*: possuem todos os nodos homogêneos, sendo que todos os nodos possuem a mesma velocidade e capacidade de processamento, além de possuírem a mesma quantidade de recursos computacionais (ex.: memória). Somente clusters com esse tipo de arquitetura possibilitam uma verdadeira análise de desempenho.

- *Arquiteturas assimétricas:* possuem nodos diferentes. Podem possuir nodos homogêneos mas com diferentes velocidades e capacidades de processamento ou nodos homogêneos com diferentes recursos de computação (ex.: memória). Arquiteturas dessa classe dificultam possíveis análises de desempenho.

Além dessas vantagens, a disponibilidade comercial de redes rápidas tem encorajado muitos experimentos no uso de clusters de SMPs a fim de se obter máquinas de alto desempenho com uma boa relação de custo/desempenho. Essas redes de interconexão oferecem tempo de latência e largura de banda comparáveis com as redes de interconexão proprietárias que são encontradas nas MPPs. A seguir, são apresentadas algumas métricas relacionadas à transferência de dados por redes de interconexão.

- *Packing time* (Tempo de Empacotamento): é o tempo gasto para tornar os dados disponíveis para o envio, incluem codificação, no caso de comunicação heterogênea, e armazenamento (cópia) dos dados para um *buffer* apropriado quando necessário.

- Latência: é o tempo que decorre desde que a mensagem é enviada até que ela se torne disponível no lado do receptor, incluindo o tempo que leva para executar as operações de envio e recebimento, e o tempo gasto em protocolos de comunicação, drivers de rede e a trasferência dos dados através do meio.

- *Bandwidth* (Largura de Banda): é calculado baseado na latência e no tamanho das mensagens, indicando a máxima taxa de transferência da rede, normalmente é medida em Mbytes/second.

As métricas latência e *bandwidth* variam de acordo com o tamanho das mensagens. Quanto maior o tamanho da mensagem maior será o tempo de latência e maior será a largura de banda, sendo que essa última para um tamanho de mensagem muito grande, tende a saturar.

A fim de garantir um melhor desempenho na comunicação, vários padrões de interconexão estão sendo desenvolvidos para conectar os nós dessas máquinas. A seguir serão caracterizados os padrões mais citados na literatura:

- Fast Ethernet

A *Switch Fast-Ethernet* garante uma latência muito menor na comunicação entre máquinas, através da emulação de uma conexão ponto-a-ponto entre todas as máquinas (é feito um "chamamento" em hardware ligando os nós da rede a cada comunicação). Placas convencionais de interconexão *Fast-Ethernet* possuem uma vazão nominal de 100 Mb/s. O fato de ser uma placa convencional implica na implementação das camadas de rede em software o que compromete a latência de forma significativa. Nas outras tecnologias de interconexão essas camadas são implementadas em hardware, o que melhora a latência da comunicação.

- ParaStation

A interface de programação apresentada pela *ParaStation* consiste de uma emulação de *sockets* UNIX e de ambientes amplamente utilizados para programação paralela, como PVM [GEI94a]. Isto permite portar uma grande quantidade de aplicações paralelas e cliente/servidor para a ParaStation. Algumas implementações inicias da ParaStation

atingiram uma latência em torno de 2 microssegundos e uma largura de banda de 15 Mbyte/s por canal de comunicação. Uma rede ParaStation utiliza uma topologia baseada em uma malha toroidal[1] de duas dimensões, mas para sistemas pequenos uma topologia em anel é suficiente.

O objetivo da ParaStation é prover uma padronizada e eficiente interface de programação no topo da rede. A rede é dedicada a aplicações paralelas e não pretende substituir LANs comuns, desta forma os protocolos padrão de LANs podem ser eliminados. Isto permite utilizar propriedades mais especializadas na rede, como protocolos ponto-a-ponto e controle da rede ao nível do usuário sem interação com o sistema operacional. O protocolo ParaStation implementa múltiplos canais lógicos de comunicação em uma ligação física. Em contraste com outras redes de alta velocidade, como a Myrinet por exemplo, na ParaStation não há custo adicional para componentes de switch central.

- Myrinet:

É um novo tipo de rede que utiliza uma tecnologia baseada em comunicação através de pacotes. As características que tornam a *Myrinet* uma rede de alto desempenho, incluem o desenvolvimento de canais robustos de comunicação com controle de fluxo, pacotes, controle de erro, baixa latência, interfaces que podem mapear a rede, rotas selecionadas, tradução de endereços da rede para essas rotas, bem como manipulação do tráfego de pacotes e software que permite comunicação direta entre os processos a nível de usuário e a rede.

A Myrinet foi originalmente desenvolvida para ser utilizada em sistemas multicomputadores (MPP's e NOW's), que consistem de uma coleção de nós de computação, cada um com sua própria memória, conectados por uma rede de troca de mensagens. Atualmente a Myrinet vem sendo utilizada em máquinas baseadas em *clusters*. Do mesmo modo que as LANs, os nós de uma máquina baseada em *clusters* utilizam uma rede Myrinet, eles enviam e recebem dados na forma de pacotes. Qualquer nodo pode enviar um pacote para qualquer outro nodo. Um pacote consiste de uma seqüência de bytes iniciando com um cabeçalho que é examinado pelos circuitos de roteamento para encaminhar o pacote através da rede. Em contraste com as LANs comuns, porém, esta rede baseada em Myrinet possui altas taxas de transferência. Uma ligação Myrinet é composta por um par de canais full-duplex que permite uma taxa de transferência de cerca de 1.28 Gbit/s cada um.

Uma rede Myrinet utiliza normalmente topologias regulares, tipicamente malhas de duas dimensões, embora ela permita a utilização de uma topologia arbitrária uma vez que um cabo Myrinet, pode conectar *hosts* entre si, ou ainda ligar uma placa a um *switch* ou ainda dois *switches* entre si. Ao contrário de uma LAN típica onde todo o tráfego de pacotes compartilha um mesmo canal físico, uma rede Myrinet com uma malha bidimensional pode ser considerada escalável, pois a capacidade dos agregados cresce com o número de nós devido ao fato de que muitos pacotes podem trafegar de forma concorrente

[1] significado geométrico: sólido gerado pela rotação de uma superfície plana fechada em torno de um eixo que não lhe seja secante.

por diferentes caminhos da rede. Uma rede Myrinet é composta de ligações full-duplex ponto-a-ponto que conectam *hosts* e *switches*. Os switches com múltiplas portas podem ser conectados por ligações para outros *switches* e para outros *hosts* em topologias variadas.

A Myrinet é uma tecnologia de chaveamento e comunicação de pacotes de alto desempenho (ela permite uma latência de cerca de 5 microssegundos) e um custo relativamente baixo que está sendo amplamente utilizada para interconectar máquinas baseadas em clusters.

- SCI (Scalable Coherent Interface)

SCI é um padrão recente que especifica um hardware e protocolo para conexão de até 64K nós em uma rede de alta velocidade com características de comunicação de alto desempenho [EIC95] [GEI94a].

O SCI define serviços de barramento oferecendo soluções distribuídas para a sua realização. O mais notável destes serviços é um espaço de endereçamento físico de 64 bits entre os nós SCI que permite operações de escrita, leitura e a criação de áreas de memória compartilhada entre os nós.

Dos 64 bits de endereçamento para a DSM (*Distributed Shared Memory*), 16 bits são utilizados para endereçar os 64 nós possíveis[2] e os restantes 48 bits para endereçamento em cada nodo. A placa SCI permite construir máquinas com características NUMA (*Non Uniform Memory Access*), uma vez que estas placas permitem acessos à memória remota (DSM) realizados pelo hardware, mas que são mais lentos que os acessos locais, o que caracteriza acessos não uniformes à memória [HWA93]. Protocolos para coerência de cache em memória compartilhada distribuída podem ser desenvolvidos para estes sistemas baseados em NUMA.

O SCI evita a limitação física dos barramentos pelo emprego de ligação unidirecional ponto a ponto. Deste modo, não há maiores dificuldades para a escalabilidade. As ligações podem ser rápidas e seu desempenho pode aumentar com a utilização de tecnologia de ponta. Tais ligações podem ser implementadas com linhas de transmissão paralela ou serial baseadas em diferentes mídias (ex.: fibra ótica). O SCI especifica uma largura de banda inicial de 1 Gbit/s para ligação serial e 1 Gbyte/s usando uma canal paralelo, ambos sobre curtas distâncias.

A construção básica de blocos SCI é através de pequenos anéis. Sistemas maiores podem ser obtidos através da criação de anéis de anéis, interconectados via SCI switches. Desta forma, além de permitir a troca de mensagens utilizando um hardware especial o SCI ainda possui a capacidade de implementar via hardware uma memória compartilhada distribuída (DSM), através de operações de escrita e leitura em regiões de memória mapeadas em memórias remotas. Isto se traduz em baixa latência, taxa na ordem de poucos microssegundos num ambiente baseado em *clusters*.

1.1 Configurações de Clusters

[2] $2^{16} = 64K$

Nesta seção serão apresentadas três configurações possíveis de clusters que foram apresentadas em [DER99]. Será adotado que essas configurações possuirão um número fixo de nodos uma vez que bastaria recursos para a inclusão de novos.

A configuração mínima se caracteriza por utilizar placas convencionais *Fast-Ethernet* além de uma *Switch Fast-Ethernet* para a interconexão dos nós da máquina. É importante ressaltar que apesar de a diferença para uma rede de estações (NOW) parecer pequena, essa *Switch* garante uma latência muito menor na comunicação entre máquinas, através da emulação de uma conexão ponto-a-ponto entre todas as máquinas (é feito um chamamento em hardware ligando os nós da rede a cada comunicação). Esse é o ponto determinante que faz com que essa máquina pertença à classe de máquinas baseadas em clusters e não à classe de redes de estações (NOW's).

Essa configuração é denominada mínima porque o uso de placas convencionais implica na implementação das camadas de rede em software o que compromete a latência de forma significativa.

A configuração básica caracteriza- se por utilizar uma rede de baixa latência para a interconexão dos nós. Essa denominação foi utilizada para representar a interconexão dos nós por placas de baixa latência e não por placas de rede convencionais. A principal diferença para a configuração mínima é que as camadas de rede são implementadas em hardware nas placas, e não em software como na configuração anterior, o que melhora a latência na comunicação.

Como nesse caso o valor de latência se aproxima consideravelmente das máquinas MPP, já se torna possível neste caso comparar as duas arquiteturas (clusters e MPP) em nível de desempenho. Como essa configuração não implementa uma memória global por hardware, como a configuração avançada (será vista adiante), a implementação de uma memória global e distribuída em software é uma possível área de pesquisa.

No caso da configuração mínima e básica, pode-se utilizar bibliotecas que implementem DSM sobre memória distribuída como a biblioteca TreadMarks [AMZ95]

A configuração avançada caracteriza-se por utilizar duas redes de interconexão distintas, uma que se utiliza de uma *Switch Fast-Ethernet* (equivalente a configuração mínima) e outra que se utiliza de placas de interconexão especiais do padrão SCI. A idéia aqui é utilizar a rede *Fast-Ethernet* para a tráfego de E/S, monitoração e gerência de recursos do sistema, liberando a rede de menor latência para o tráfego exclusivo de mensagens das aplicações paralelas.

As placas SCI são ligadas entre si por conexões ponto-a-ponto e para um pequeno número de nós (2 a 10) se recomenda a ligação em anel. É importante ressaltar que a principal diferença em nível de arquitetura da máquina é que a placa SCI implementa também uma memória global em hardware, dando uma maior versatilidade na programação

desta configuração. A latência da placa SCI é equivalente as placas usadas na configuração básica (poucos microssegundos), pois também implementa as camadas de rede em hardware.

A programação de máquinas clusters pode ser feita com bibliotecas padrão para a programação paralela, como PVM, que se encontram disponíveis para o sistema operacional Linux e são gratuitas, sendo que outra possibilidade é a programação utilizando o mecanismo de *Sockets* [DUM95] disponíveis no Linux. Ambas trabalham com o modelo de comunicação de troca de mensagens que se adapta bem ao caso da configuração mínima e básica uma vez que essas possuem memória distribuída. No caso da configuração avançada, há mais possibilidades de programação, uma vez que além da memória distribuída há a possibilidade de existência de uma memória global entre as máquinas.

1.2 O Cluster de Alto Desempenho da UFRGS

O Grupo de Processamento Paralelo e Distribuído da UFRGS possui como plataforma de execução para aplicações paralelas um cluster homogêneo formado por 4 nodos. Cada nodo do cluster é um Dual Pentium Pro (2-way SMP) com 64M de memória RAM e clock de 200Mhz. Estes nodos estão interconectados por duas redes de comunicação: uma rede Fast Ethernet e outra rede Myrinet. Além disso existem algumas máquinas que possuem a função de console e/ou servidor NFS. O sistema operacional é Linux, com kernel 2.2.1 e compilador C++ (gcc) versão 2.91.60 (egcs-1.1.1). Estão instaladas as bibliotecas PVM 3.4 e duas implementações de MPI, LAM 6.1, da Universidade de Ohio [OHI96], e MPICH 1.1.2, do Laboratório de MCS [GRO96].

Observações sobre o cluster da UFRGS:

- A rede Myrinet está isolada, sendo acessível somente para a execução de aplicações paralelas de dentro do cluster.

- Para usar a rede Fast Ethernet, basta citar o nome da maquina ou o IP nos programas. Para usar a Myrinet, deve-se utilizar o segundo nome da maquina ou o segundo IP.

- A máquina meyer encontra-se no conectada ao switch Fast-Ethernet, mas possui placa Ethernet.

A Tabela 1.1 mostra a relação de todas as máquinas pertencentes ao ambiente do cluster, sendo que as máquinas meyer, scliar e ostermann possuem a função de console (terminais de acesso ao cluster) e as máquinas dionélio, verissimo, quintana e euclides constituem os nodos do cluster, onde são executadas as aplicações paralelas.

Tabela 1.1 Características das máquinas pertencentes ao cluster.

Máquina	Arquitetura	Características Especiais
Meyer	Pentium PRO Single	Console. Não está conectado à rede Myrinet
Scliar	Pentium PRO Single	Servidor NFS do cluster
Ostermann	Pentium II	Console
Dionelio	Dual Pentium PRO 200	Nodo do cluster

Verissimo	Dual Pentium PRO 200	Nodo do cluster
Quintana	Dual Pentium PRO 200	Nodo do cluster
Euclides	Dual Pentium PRO 200	Nodo do cluster

As Tabelas 1.2 e 1.3 relacionam os nomes das máquinas pertencentes ao ambiente do cluster com seus respectivos números IPs e seus nomes no servidor NFS. Na Tabela 1.2 estão as máquinas que não fazem parte do cluster propriamente dito e possuem funções de consoles ou de servidor NFS das redes Fast Ethernet e Myrinet (caso da máquina scliar, meyer e ostermann). Essas máquinas que possuêm função de consoles ou servidores NFS recebem um tratamento especial nas configurações, além de possuírem processadores diferentes dos nós do cluster, logo não são contadas como nós da máquina paralela. Como os consoles são responsáveis por toda a E/S da máquina paralela e ainda têm funções de carga de programas e de monitoração, elas já sofrem uma carga considerável. Isso naturalmente não impede que elas sejam usadas no processamento de aplicações paralelas. A inclusão dessas máquinas no processamento de aplicações paralelas, continuaria a deixar o cluster homogêneo (todas as máquinas reconhecem o mesmo conjunto de instruções) mas esse se tornaria assimétrico (não possuem as mesmas características de processamento), além de que a sobrecarga que essas máquinas recebem tem que ser considerada, o que dificultaria uma possível análise de desempenho. Na Tabela 1.3 estão os dados referentes as máquinas que fazem parte do cluster propriamente dito.

Tabela 1.2 Endereços IPs das máquinas consoles.

Máquina	IP/NFS (Fast-Ethernet)	IP/NFS (Myrinet)
Meyer	143.54.7.130/meyer	não está conectada
Scliar	143.54.7.131/ scliar	192.168.1.1/ mscliar ou scliar_m
Ostermann	143.54.7.137/ ostermann	192.168.1.7/ mostermann ou ostermann_m

Tabela 1.3 Endereços IPs das máquinas pertencentes ao cluster.

Máquina	IP/NFS (Fast-Ethernet)	IP/NFS (Myrinet)
Verissimo	143.54.7.132/ verissimo	192.168.1.2/ mverissimo ou verissimo_m
Quintana	143.54.7.133/ quintana	192.168.1.3/ mquintana ou quintana_m
Dionelio	143.54.7.134/ dionelio	192.168.1.4/ mdionelio ou dionelio_m
Euclides	143.54.7.135/ euclides	Meuclides ou euclides_m

A Figura 1.1 apresenta o esquema de conexão dos nodos das máquinas nas redes Fast Ethernet e Myrinet. Como cada máquina ficou com duas interfaces de rede, foram configurados endereços IP diferentes em cada uma delas. Desse modo, pode-se alternadamente, executar programas de teste em uma ou outra rede, bastando escolher os endereços IP de origem e destino das conexões. Com esta configuração a rede Myrinet ficou isolada da rede externa, não sendo possível que trafeguem por ela dados originários de fora do cluster. A Figura 1.2 apresenta uma foto do cluster de alto desempenho utilizado pelo Grupo de Processamento Paralelo e Distribuído da UFRGS.

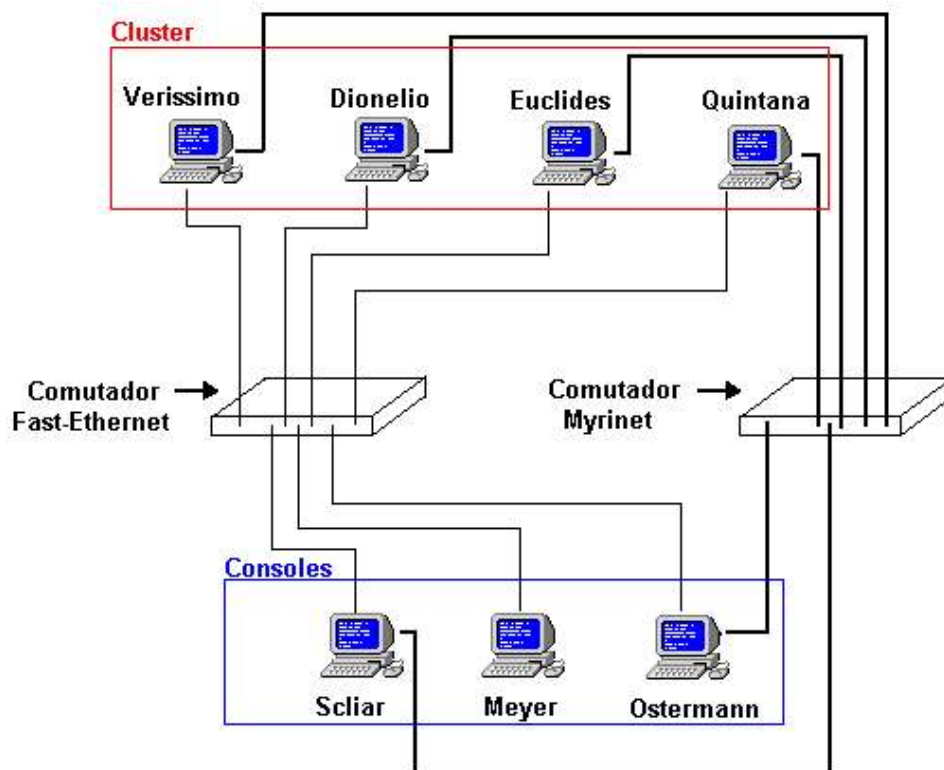



Figura 1.1 Atual configuração do cluster de alto desempenho da UFRGS

Figura 1.2 Torre do cluster composto por 4 máquinas

1.3 Objetivo e organização desse relatório

Este relatório foi desenvolvido com a finalidade de apresentar métodos de construção de programas paralelos, utilizando-se os recursos disponíveis pelo Grupo de Processamento Paralelo e Distribuído da UFRGS. No primeiro capítulo foram apresentadas as características e conceitos dos clusters, seguida de uma breve descrição dos recursos disponíveis pelo grupo.

No segundo capítulo são descritas três ferramentas, que se encontram disponíveis no cluster do Grupo de Processamento Paralelo e Distribuído da UFRGS. Essas ferramentas (PVM, MPI, DPC++) possibilitam a construção de programas paralelos e distribuídos, sendo feita uma comparação entre as características dessas ferramentas na construção dos mesmos.

No capítulo três é feita uma introdução do ambiente necessário para se executar essas ferramentas, dando-se ênfase à ferramenta DPC++. Nesse capítulo é feita uma introdução ao sistema operacional Linux, às linguagens orientadas a objetos e à linguagem C++. Esses assuntos são abordados por estarem diretamente ligados ao modelo DPC++.

No quarto capítulo é feita uma abordagem do ambiente de compilação DPC++ como uma ferramenta que possibilita a construção de programas paralelos que podem ser executados no cluster da UFRGS. Nesse capítulo é discutido o modelo de distribuição adotado pelo DPC++, uma apresentação da linguagem DPC++, que inclui diretivas (comandos) do DPC++ e restrições quanto ao C++. Também é feita uma descrição do ambiente de compilação (módulos constituintes), instalação do DPC++, configuração do ambiente e como definir aplicações utilizando-se o DPC++. No quinto capítulo são apresentados alguns programas escritos em DPC++ a fim de exemplificar ao iniciando no ambiente como são definidos os programas na prática.

2 Ferramentas de Programação em Clusters

Um ambiente de programação de alto desempenho é constituído de vários processadores. Esses processadores podem estar em uma única máquina ou distribuídos em várias máquinas. O processamento nessas máquinas são ditos: paralelo e distribuído, respectivamente. Em ambos os tipos de processamento, há a necessidade de troca de informações entre os processadores para que eles cooperem para o desenvolvimento ou resolução de um problema.

Quando se tem todos os processadores em uma única máquina, em geral, a comunicação se dá através da memória principal, global, a qual é compartilhada por todos os processadores.

Quando se tem os processadores distribuídos em estações de trabalho, há a necessidade de comunicação entre processadores. Usualmente, cada processador opera sobre um conjunto próprio de dados armazenados em uma memória local. Eles se comunicam através de ferramentas e bibliotecas de comunicação. Nesse contexto cada processador se constitui um nodo da rede.

Atualmente, diferentes software de comunicação são usados para permitir comunicação e disponibilização de paralelismo através de uma rede. As ferramentas podem ser divididas em bibliotecas de comunicação e linguagens paralelas. Como exemplos de bibliotecas de comunicação, tem-se o PVM, MPI e Athapascam e como exemplos de linguagens paralelas tem-se o DPC++. Inicialmente esses software foram projetados para sistemas distribuídos, com memória distribuída. A seguir será feita uma introdução a respeito das ferramentas PVM, MPI e DPC++.

2.1 PVM - Parallel Virtual Machine

PVM [GEI94] é uma biblioteca de comunicação que emula computação concorrente heterogênea de propósitos gerais em computadores interconectados de variadas arquiteturas A idéia do PVM é montar uma máquina virtual de *n* processadores e usá-los para enviar tarefas e receber os resultados. Essa coleção de computadores (máquina virtual) pode ser usada de uma maneira cooperativa para computação concorrente ou paralela. Por computação concorrente entende-se um conjunto de processos que concorrem entre si a fim de obter recursos computacionais (ex.: processador, memória, periféricos,..). A computação concorrente pode ser praticada sobre um único processador, não necessitando obrigatoriamente mais de um processador. O conceito de computação paralela se refere a processar um ou mais processos simultaneamente, logo esse conceito exige que haja no mínimo dois processadores.

O ambiente PVM é composto de três partes principais. A primeira parte é o *console* que é usado para montar a máquina paralela virtual, através de primitivas próprias. O ideal é criar essa máquina uma única vez para várias aplicações, ou seja, a máquina estará disponível enquanto o programador não a destruir. A segunda parte é um *daemon*, que é um programa que roda em todos os nodos constituintes da máquina virtual e é responsável pelo controle das tarefas que estão sendo executadas nesses nodos. A ação de inserir uma máquina no ambiente virtual nada mais é do que iniciar um processo *daemon* na mesma, sendo que a remoção de uma máquina nada mais é do que matar o referido processo. A terceira parte do ambiente PVM é uma biblioteca das rotinas de interface, que contém um conjunto de primitivas que são necessárias para a cooperação

entre tarefas de uma aplicação. Como exemplo de primitivas temos: *pvm_send()*, *pvm_receive(), pvm_spawn*, etc. Essa biblioteca dispõe de recursos que possibilitam a manipulação do ambiente virtual, inclusive em tempo de execução, como retirar e inserir processadores, criar novos processos, matar processos, enviar mensagens para vários processos ao mesmo tempo, e inúmeras outras possibilidades. Uma observação importante quanto a manipulação da máquina virtual (adicionar e remover máquinas) em tempo de execução, é que, além de não ser prático, possui um custo computacional muito grande.

O PVM possui como característica ser de fácil portabilidade entre arquiteturas diferentes e sistemas operacionais diferentes, tanto que ele pode ser usado até sobre o sistema operacional Windows NT. O PVM, por exemplo, é tão heterogêneo que não só funciona em várias arquiteturas, mas como se pode rodar uma mesma aplicação nelas. Exemplo: pode-se disparar uma aplicação usando como nodos as máquinas *scliar*, *verissimo* (Linux), *poncho* (Solaris) e *pala* (SUN4), todas disponíveis no laboratório do Grupo Processamento Paralelo e Distribuído da UFRGS. Algumas poucas diferenças na hora de compilar as aplicações são necessárias para se portar para outras arquiteturas.

Como facilidades para o programador (em relação a paralelismo implícito ou explícito) o PVM não possui nenhuma, sendo necessário que o programador explicite a criação de tarefas, sua comunicação e destruição, sendo possível, inclusive, explicitar em qual processador uma tarefa deve ser iniciada. Esse fato dificulta um pouco a adaptação por parte do programador, exigindo desse um estudo rigoroso de suas primitivas, além de um bom embasamento na área de projeto de algoritmos paralelos e distribuídos. Por possuir apenas uma biblioteca que disponibiliza métodos que permitem o paralelismo, o PVM não gera código para o programador, sendo que este tem que se encarregar de dividir suas tarefas em processos e dispará-los nos nodos. O programador pode, se desejar, fazer sua própria rotina de escalonamento e chamá-la de dentro da primitiva *pvm_spawn()*.

O PVM não possibilita a modelagem em um nível mais alto de abstração (ex.: necessita-se ater a detalhes de paralelismo e comunicação), o que torna mais complexo a modelagem de problemas e respectiva definição de aplicações. Quanto a programação de aplicações, o PVM permite que os programas possam ser escritos nas linguagens C, C++ e Fortran, sendo uma grande vantagem a reutilização de código previamente escrito. Para a definição de aplicações, o cabeçalho da biblioteca que contém as rotinas de interface do PVM deve estar incluso na aplicação. Verifica-se que iniciar processos dinamicamente, ou seja, de dentro de outro processo, gera um *overhead* muito grande além de confundir um pouco a lógica de programação. As trocas de dados são feitas por trocas de mensagens, sendo que a comunicação entre nodos no PVM é realizada pelos processos do *daemon* que são responsáveis por receberem as mensagens e entregá-las para as tarefas locais. Um exemplo de código em PVM pode ser visto no apêndice deste trabalho. As versões mais novas do PVM suportam *threads*, que permite explorar máquinas com mais de um processador.

2.2 MPI - Message Passing Interface

MPI [MPI94], normalmente referida como MPI standard (padrão), é um esforço para melhorar a disponibilidade de eficiência e portabilidade do software por garantir as necessidades de aplicações paralelas e distribuídas. O MPI standard define um padrão de troca de mensagens onde cada fabricante é livre para implementar as rotinas, já com sintaxe definida, utilizando características exclusivas de sua arquitetura.

Atualmente existem muitas implementações de MPI (como por exemplo MPICH [GRO96], LAM [OHI96]) de diferentes instituições e fabricantes. Todas as implementações mantêm a mesma API para o usuário, contudo elas apresentam algumas diferenças nas opções de utilização (ex.: flags de programação) que resultam em diferentes desempenhos.

Inicialmente com o objetivo de otimizar a comunicação, o gerenciamento dinâmico de processos e processadores não era suportado, ao contrário do PVM. A princípio esta restrição pode parecer desfavorável, mas os programas escritos em MPI tendem a ser mais eficientes pelo fato de não haver *overhead* na carga de processos em tempo de execução. O MPI, no entanto, por não prover gerenciamento dinâmico de processos, deixa de ser atraente para algumas aplicações, tipicamente para aquelas onde cada processador é responsável por uma tarefa específica. Para usar o MPI nesses casos, o que se fazia era um código bastante grande com vários comandos **if**. Esses processos usavam muitos recursos de hardware e sua inicialização era custosa. [MPI97]

Uma característica marcante no MPI é que não existem "processos" como no PVM, existe sim, um único processo que pode ser rodado em várias máquinas (chamadas de nodos) já previamente montadas na "máquina virtual" e não modificável em tempo de execução. Logo se percebe que esta técnica se beneficia em muito de programas paralelos simétricos, onde todos os participantes executam o mesmo trecho de código, só que em porções de dados diferenciadas. Como pode se observar, MPI é mais indicado para problemas que são facilmente lidados por arquiteturas SIMD (Single Instruction, Multiple Data), onde se dispara o mesmo processo em todos os nodos. A versao 2 do MPI já possui um MPI_Spawn(), para disparar processos, permitindo disparar processos não simétricos em máquinas diferentes.

Por ser apenas uma biblioteca, o MPI não gera código para o programador. Trabalha-se com bibliotecas, incorporando-as ao código do programa e executando suas primitivas. A portabilidade e a eficiência são pontos fortes do MPI, que pode ser verificado pelas diferentes implementações mas com a mesma API para o desenvolvimento de aplicações. O MPI permite que os programas possam ser escritos nas linguagens C, C++ e Fortran, possibilitando reutilização de código previamente escrito.

Em relação à facilidade de adaptação por parte do programador (exigência de conhecimentos específicos), o MPI necessita um estudo rigoroso de suas primitivas, precisa-se explicitar a criação das tarefas, suas comunicações e destruição. Esse fato dificulta a modelagem de problemas em um nível mais alto de abstração, necessitando-se ater a detalhes do paralelismo em si.

A programação com o MPI é mais simples e mais legível que a do PVM. O simples fato de no PVM existirem *n* códigos diferentes de uma mesma aplicação e no MPI haver apenas um, já o torna mais atraente. MPI também trouxe alguma comunicação implícita, como o MPI_REDUCE. Neste caso, define-se uma variável e determina-se que ela será reduzida de todos os processos por uma operação matemática. Como exemplo, podemos citar um programa que calcule o PI em que cada processo calcula uma parte e o resultado final é a soma de todas as partes, no caso um MPI_REDUCE pela soma. O efeito é o mesmo do que se cada processo enviasse uma mensagem com a sua parte ao nodo raiz (nodo 0), para que este faça a soma. As versões mais recentes do PVM também já trazem este tipo de operações.

O mecanismo utilizado para trocar dados é o método de troca de mensagens sendo que fica sob a responsabilidade do programador a introdução dos comandos para que essa troca se realize. Em relação ao compartilhamento de memória (memória comum), existem implementações de bibliotecas MPI para arquiteturas com memória compartilhada. As versões mais novas do MPI (MPI-2) suportam *threads*, que permite explorar máquinas com mais de um processador.

2.3 DPC++ - Distributed Processing in C++

Além das bibliotecas mencionadas anteriormente existe, atualmente, na UFRGS, um avançado projeto de ambiente de compilação chamado DPC++ (Distributed Processing in C++). Inicialmente projetado para sistemas distribuídos com memória distribuída, o DPC++, atualmente, está implementado diretamente sobre sockets de UNIX, mas logo poderá ser facilmente portado para outros sistemas, graças a um outro projeto chamado DECK (Distributed Executive Communication Kernel) que é uma camada de execução de propósitos gerais dedicada à aplicações paralelas. Ela provê ao usuário uma API com um conjunto de primitivas que lidam com threads, operações de definição de comunicações e de sincronização, da mesma forma que um número básico de serviços, como nomeação, empacontamento/desempacotamento, carregamento balanceado de tarefas, etc..., necessárias pela maioria das aplicações paralelas.

DECK é um ambiente dedicado para clusters formados por estações de trabalho homogêneas, interconectadas via uma interface LAN (FastEthernet, Myrinet,etc.), e faz uso de um mecanismo de comunicação do tipo sockets de UNIX em conjunto com uma biblioteca “Pthreads-compliant” para prover multiprogramação.

O DPC++ possui como método de troca de dados o mecanismo de troca de mensagens, só que isso é feito de uma maneira implícita (nunca o programador fará uso de uma primitiva *send* de forma explícita. Isso se deve ao fato de que quando o programador cria um objeto distribuído, esta criação, o envio de parâmetros e, posteriormente o retorno, é feito com troca de mensagens. Os serviços *send* e *receive* não podem ser usados pelo programador).

O DPC++ ao contrário do PVM e MPI é uma linguagem de programação, possuindo um compilador próprio que necessita, inclusive de uma biblioteca de troca de mensagens para funcionar, podendo ser até mesmo o PVM ou o MPI. Inicialmente o DPC++ foi implementado em sockets, com um protótipo em PVM, mas atualmente, está sendo migrado para o DECK. Então,

para uma possível comparação de desempenho entre PVM, MPI e DPC++ ser válida, deve-se especificar qual a biblioteca atualmente usada no DPC++.

Em relação a facilidades para o programador (paralelismo implícito, explícito, ambos) pode-se dizer que o DPC++ não possui paralelismo implícito, pois o programador precisa dividir suas tarefas em objetos. Por outro lado, se tem uma grande diferença em relação ao PVM e MPI: a troca de mensagens entre os processos da aplicação é de forma implícita, possibilitando uma modelagem em um nível mais alto de abstração (o programador não precisa se ater a detalhes da comunicação em si). Pode-se afirmar que o DPC++ extrai paralelismo, pois, a rigor, o programador não está dividindo seu problema em tarefas para serem executadas em outros nodos. Ele o está dividindo em objetos, da mesma forma que é feita em C++[0]. Objetos que são criados, resolvem algo e retornam suas respostas. É aí que o DPC entra, pois:

- Criar objeto: significa instanciar um objeto de uma determinada classe em um nodo X, significa enviar uma mensagem para o nodo, pedindo que torne ativo o objeto;

- Executar método X do objeto Y: significa enviar uma mensagem com os parâmetros necessários para o nodo onde o objeto Y se encontra instanciado, sendo que este nodo invocará o método solicitado enviando uma mensagem ao nodo solicitante, com a resposta;

- Destruir um objeto: significa enviar uma mensagem ao nodo hospedeiro deste objeto, para torna-lo inativo.

Obs.: As considerações acima se referem ao novo modelo do DPC++ proposto e não ao atualmente implementado).

A princípio o DPC++ não possui mecanismos de tolerância a falhas, mas se espera que em uma versão futura o DPC++ possua tais mecanismos.

Atualmente o DPC++ não é de fácil portabilidade entre arquiteturas diferentes e sistemas operacionais diferentes, pois está implementado sobre o mecanismo de sockets de UNIX. Assim que o DPC++ estiver fazendo uso dos serviços de comunicação do DECK, poderá ser portado para os sistemas que o DECK permitir, como o DPC++ é dependente da biblioteca de comunicação que faz uso, a portabilidade depende em princípio da biblioteca de comunicação.

Uma possível comparação de desempenho entre PVM, DPC++ e MPI não é possível uma vez que DPC++ não é uma biblioteca e a troca de mensagens não é responsabilidade dele. O DPC++ necessita de uma biblioteca de comunicação, logo, para fazer uma comparação entre PVM, MPI e DPC++, dever-se-ía fazer uma comparação entre PVM, MPI e sockets (atual biblioteca de comunicação utilizado pelo DPC++). Como a tendência é de que o DPC++ seja implementado sobre o DECK, poder-se-ía fazer uma comparação entre PVM, MPI e DECK.

[0] Uma observação importante é que a linguagem DPC++ é derivada do C++, possibilitando uma possível reutilização de código previamente escrito.

2.4 Considerações finais

Existem alguns ambientes disponíveis para a programação no cluster da UFRGS: PVM, MPI, DPC++, Athapascam. Sendo que é feito uma comparação das características dos três primeiros ambientes de programação paralela. Nesse trabalho está sendo dada especial atenção ao ambiente DPC++, por fazer uso do paradigma de orientação a objetos, por ser uma extensão da linguagem C++, e não exigir um grande esforço do programador para garantir a comunicação entre os processos paralelos uma vez que essa comunicação é implícita, bastando ao programador modelar o problema no paradigma de orientação a objetos. Além dos motivos expostos anteriormente, cabe ressaltar que, o DPC++ é um projeto do Grupo de Processamento Paralelo e Distribuído da UFRGS.

A biblioteca PVM e todas as implementações de MPI foram inicialmente desenvolvidas sobre o protocolo TCP/IP. Conseqüentemente seu simples uso em uma arquitetura baseada em *cluster*, com rápidas redes de interconexão, resulta em uma desempenho muito aquém da máxima desempenho oferecida por essas redes. A alta taxa de tempo gasta na comunicação e na manipulação de dados são os responsáveis por esse fato. Por causa disso, alguns melhoramentos estão sendo desenvolvidos para melhor adaptar essas camadas de comunicação para esse novo hardware (MPI-FM [LAU97], PULC [WAR98], BIP [PRY98]). Esses trabalhos têm por objetivo evitar o protocolo TCP/IP e interagir diretamente com a interface da rede, fornecendo melhores taxas de tempos de latência e largura de banda.

3 Tópicos Relacionados ao Ambiente DPC++

Nesse capítulo será tratado sobre vários tópicos relacionados com o ambiente *Distributed Processing in C++* (DPC++). Será feita um introdução ao sistema operacional Linux que é atualmente o sistema operacional na qual a atual versão do DPC++ é executada. Também será feita um introdução à linguagem C++, pois a linguagem DPC++ é derivada do C++, e também serão mostradas as características de linguagens baseadas no paradigma de orientação a objetos, uma vez que a linguagem DPC++ foi desenvolvida sob esse paradigma.

3.1 O sistema operacional Linux

O Linux é um sistema operacional UNIX multitarefa, multiusuário e multiprocessado, desenvolvido há poucos anos graças aos esforços coletivos da comunidade tecnológica e, em especial, de seu idealizador, Linus Torvalds. O Linux foi primeiramente desenvolvido para PCs baseados em 386/486/Pentium, mas atualmente também roda em computadores Alpha da Compaq, Sparcs da SUN, máquinas M68000 (semelhantes a Atari e Amiga), MIPS e PowerPCs.

O Linux é um clone UNIX de distribuição livre para PCs, sendo que sua implementação é independente da especificação POSIX, com a qual todas as versões do UNIX padrão (true UNIX) estão convencionadas. Sendo um sistema baseado no padrão POSIX, desfruta as vantagens do UNIX para trabalho e gerenciamento seguro de grandes redes e programação, mas também é ideal para uso doméstico, controle administrativo de pequenas empresas e até para entretenimento.

O Linux foi escrito inteiramente do nada, logo não há código proprietário em seu interior, sendo que o sistema operacional encontra-se disponível na forma de código objeto, bem como em código fonte. Uma característica importante do Linux é que esse pode ser livremente distribuído nos termos da GNU (*General Public License*).

O Linux possui como características ótimo desempenho, estabilidade, segurança e multiplicidade de recursos, além de todas as características que se pode esperar de um UNIX moderno, incluindo:

- Multitarefa real
- Memória virtual
- Biblioteca compartilhada
- "Demand loading"
- Gerenciamento de memória próprio
- Executáveis "copy-on-write" compartilhados
- Rede TCP/IP (incluindo SLIP/PPP/ISDN)
- X Windows

A maioria dos programas rodando em Linux são freeware genéricos para UNIX, muitos provenientes do projeto GNU. A variedade de programas disponíveis no mercado é

extremamente grande, incluindo até sistemas gerenciadores de bancos de dados SQL e emuladores de terminais remotos. O fato de seus programas serem de livre distribuição reduz os custos de implementação e uso a quase nada.

3.1.1 História do Linux

Em agosto de 1991, na Finlândia, um jovem de 21 anos de idade chamado Linus Torvalds iniciou o projeto do Linux. O estudante universitário desejava desenvolver uma versão do Unix que rodasse em micros PC AT e compatíveis, mas que fosse diferente dos sistemas Unix já existentes, cujo preço era exorbitante para o usuário comum.

Linus chegou a divulgar a idéia num newsgroup de que participava e embalado pelo projeto, programou sozinho a primeira versão do kernel do Linux (núcleo do sistema operacional). Linus Torvalds se inspirou em Andy Tanembaum, criador do Minix, outro sistema operacional Unix, do qual Linus era usuário. A nova criação foi batizada com o nome de Linux, vocábulo que resultou da fusão de Unix com o primeiro nome de seu criador, Linus.

Depois de finalizar o kernel, Linus deu ao seu projeto o rumo que desencadeou seu grande sucesso: passou a distribuir o código fonte do kernel pela Internet, para que outros programadores pudessem aprimorar o sistema. Assim, várias empresas e programadores de todo o planeta contribuíram com seus conhecimentos para melhorar e fazer do Linux o sistema operacional potente e diversificado que é hoje. Esse foi o segredo: trabalho cooperativo e voluntário. Linus distribuiu seu trabalho sem cobrar nada e em troca, exigiu que os outros programadores envolvidos no projeto fizessem o mesmo. Por isso o Linux é gratuito.

Por causa da abertura do código fonte ao mundo, não existe uma, mas muitas distribuições do Linux no mercado. Todas tem características especiais que as diferenciam entre si. Na verdade, não existe "o Linux", existem "os Linux". Mas apesar de singulares, todas essas versões são compatíveis, porque utilizam o mesmo kernel.

Essa parte delicada do sistema operacional só é atualizada por um grupo restrito de experts em Linux, dentre os quais está o próprio Linus Torvalds. Essa parte do sistema é tão importante que as novas versões do kernel só podem ser distribuídas depois de passarem pelo aval de Linus.

As principais versões disponíveis são: Slackware Linux, Debian Linux, Open Linux, LinuxWare e Red Hat Linux.

A grande maioria das versões está licenciada sob a licença GNU (Licença Geral Pública). Isso garante a livre distribuição e utilização do software, bem como o acesso, estudo e utilização dos códigos fontes. Há apenas um encargo: quem se utilizar do código licenciado sob a GNU, precisa distribuir seu trabalho sob essa mesma licença.

Praticamente, todos os softwares para Linux são completos e de livre distribuição, registrados nos termos da Licença Geral Publica.

A GNU representa uma nova tendência no setor da informática, popularizando a utilização de programas, cooperando para o desenvolvimento de empresas e reunindo o trabalho conjunto dos profissionais de informática para tomar o desenvolvimento tecnológico nesse ramo muito mais democrático, acessível e universal.

3.1.2 Comandos Básicos do Linux

Comandos em UNIX possuem algumas características particulares. Eles podem ser controlados por opções e devem ser digitados em letras minúsculas. A seguir é mostrada uma lista dos principais comandos do sistema operacional Linux[0]. Para obter informações mais detalhadas sobre o uso dos comando veja o comando **man**.

cat - Oficialmente usado para concatenar arquivos. Também usado para exibir todo o conteúdo de um arquivo de uma só vez, sem pausa.

Sintaxe: **cat** < arquivo1 > < arquivo2 >... < arquivo n >,

onde <arquivo1> até <arquivo n> são os arquivos a serem mostrados. **cat** lê cada arquivo em seqüência e exibe-o na saída padrão (tela do vídeo). Deste modo, a linha de comando:

cat < arquivo >

exibirá o arquivo em seu terminal; e a linha de comando :

cat < arquivo1 > < arquivo2 > **>** < destino >

concatenará <arquivo1> e <arquivo2>, e escreverá o resultado em <destino> . O símbolo "**>**", usado para redirecionar a saída para um arquivo, tem caráter destrutivo; em outras palavras, o comando acima escreverá por cima do conteúdo de < destino >. Se, ao invés disso, você redirecionar com o símbolo "**>>**", a saída será adicionada a < destino >, ao invés de escrever por cima de seu conteúdo.

cd – Muda o diretório de trabalho corrente.

Sintaxe : **cd** < diretório >

onde <diretório> é o nome do diretório para o qual você deseja mudar. O símbolo "." refere-se ao diretório corrente e o símbolo ".." refere-se ao "diretório-pai". Para mover para um "diretório-pai", ou seja, um diretório acima do que você está,

use o comando :

cd ..

[0] As Informações sobre os comandos Linux foram obtidas na seguinte fonte:
Universidade Federal de Goiás, Instituto de Informática, home-page do Projeto de Apoio ao Usuário Internet (http://www.dei.ufg.br/~apoio/unix.html)

Obs.: Note o espaço entre "cd" e ".." .

Você também pode usar nomes-de-caminho (*pathnames*) como argumento para o comando **cd**. Por exemplo :

cd /diretorio1/diretorio2

o posicionará diretamente em "diretório2". O uso de **cd** sem nenhum argumento fará com que você retorne para o seu "*home-directory*" .

chgrp - Modifica o grupo de um arquivo ou diretório.

Sintaxe: **chgrp** [-f] [-h] [-R] gid <nome-do-arquivo>

chgrp modifica o identificador de grupo (*group ID*, gid) dos arquivos passados como argumentos. "gid" pode ser um número decimal especificando o *group id*, ou um nome de grupo encontrado no arquivo "/etc/group". Você deve ser o proprietário do arquivo, ou o *root* , super-usuário, para que possa utilizar esse comando. Algumas opções:

-f: Essa opção não reporta erros
-h: Se o arquivo for um *link* simbólico, essa opção modifica o grupo do *link* simbólico. Sem essa opção, o grupo do arquivo referenciado pelo *link* simbólico é modificado.
-R: Essa opção é recursiva. **chgrp** percorre o diretório e os subdiretórios, modificando o "gid" à medida em que prossegue.

chmod - Modifica as permissões de um arquivo ou diretório. Você deve ser o proprietário de um arquivo ou diretório, ou ter acesso ao *root*, para modificar as suas permissões.

Sintaxe : **chmod** <permissões> <nome>

onde:

<permissões> - indica as permissões a serem modificadas;
<nome> - indica o nome do arquivo ou diretório cujas permissões serão afetadas.

As permissões podem ser especificadas de várias maneiras. Aqui está uma das formas mais simples de se realizar essa operação :

1- Usa-se uma ou mais letras indicando os usuários envolvidos:

. u (para o usuário)
. g (para o grupo)
. o (para "outros")

. a (para todas as categorias acima)

2- Indica-se se as permissões serão adicionadas (+) ou removidas (-).

3- Usa-se uma ou mais letras indicando as permissões envolvidas :

. r (para *read*) (ler)
. w (para *write*) (escrever)
. x (para *execute*) (executar)

Exemplo: No exemplo a seguir, a permissão de escrita *write* é adicionada ao diretório "dir1" para usuários pertencentes ao mesmo grupo. (Portanto, o argumento <permissões> é g+w e o argumento <nome_do_arquivo> é dir1).

```
$ ls -l dir1
drwxr-xr-x 3 dir1      1024 Feb 10 11:15 dir1
$ chmod g+w dir1
$ ls -l dir1
drwxrwxr-x  3 dir1     1024 Feb 10 11:17 dir1
$
```

Figura 3.1 Exemplo de utilização do comando **chmod**

Como pode-se verificar, o hífen (-) no conjunto de caracteres para grupo foi modificado para "w" como resultado deste comando. Quando se cria um novo arquivo ou diretório, o sistema associa permissões automaticamente. Geralmente, a configuração "default" (assumida) para os novos arquivos é:

- r w - r - - r - -

Assumindo como ordem de leitura da esquerda para a direita, o primeiro atributo (1) é o atributo que diferencia um arquivo de um diretório. No caso de ser um arquivo esse atributo não possui nenhum valor. Os três atributos seguintes (2, 3 e 4) são relativos às permissões do usuário, os atributos das posições 5, 6, 7 são de propriedade do grupo ao qual o usuário faz parte e os três últimos (8, 9, 10) são de propriedade das outras pessoas que não são nem o usuário e nem pertencente ao grupo do usuário. É importante observar que os atributos referentes à posição 4, 7 e 10 são atributos de execução. Se esses atributos possuírem o valor "x", então ele representa um arquivo executável, sendo que sua execução é restrita aos que possuírem o respectivo atributo de execução ativado. Uma observação importante é que nem todos os arquivos que contêm o valor "x" nessas posições de atributos são executáveis, mas todos os arquivos executáveis devem conter numa dessas posições de atributos o valor "x". Isso se deve ao fato de o usuário ser capaz de mudar o atributo de um arquivo do tipo texto. Esse arquivo poderá possuir o atributo de executável, apesar de ser um arquivo texto e assim sendo, o computador não será capaz de executá-lo.

Geralmente, a configuração "default" (assumida) para os novos diretórios é:

d r w x r - x r - x

A leitura dos atributos deve ser feita da mesma maneira como é feita para os arquivos (da esquerda para a direita), sendo que a única diferença é que o primeiro atributo possui o valor "d" de diretório. Os atributos seguintes possuem a mesma interpretação dos atributos de um arquivo.

chown - Modifica o proprietário de um arquivo ou diretório.

Sintaxe: **chown** [-fhR] <proprietário> <nome-do-arquivo>

O argumento <proprietário> especifica o novo proprietário do arquivo. Este argumento deve ser ou um número decimal especificando o *userid*, número de identificação do usuário, ou um "*login name*" encontrado no arquivo "/etc/passwd". Somente o proprietário do arquivo ou o *root*, super-usuário, podem modificar o proprietário deste arquivo. Algumas opções:

-f: Esta opção não reporta erros.
-h: se o arquivo for um *link* simbólico, esta opção modifica o proprietário do *link* simbólico. Sem esta opção, o proprietário do arquivo referenciado pelo *link* simbólico é modificado.
-R: Essa opção é recursiva. **chown** percorre o diretório e os subdiretórios, modificando as propriedades à medida em que prossegue.

cp – Copia arquivos para um outro arquivo ou diretório.

Sintaxe: **cp** <arquivo1> <arquivo2> ... <arquivo n> <destino>

onde <arquivo1> até <arquivo n> são os arquivos a serem copiados, e <destino> é o arquivo ou o diretório para onde os arquivos serão copiados. O(s) arquivo(s) fonte (s) e o <destino> não podem ter o mesmo nome. Se o arquivo-destino não existe, **cp** criará um arquivo com o nome especificado em <destino>. Se o arquivo-destino já existia antes e não for um diretório, **cp** escreverá o novo conteúdo por cima do antigo.

Exemplo : $ **cp** -r temp temp1

Este comando copia todos os arquivos e subdiretórios dentro do diretório temp para um novo diretório temp1. Esta é uma cópia recursiva, como designado pela opcão -r. Se você tentar copiar um diretório sem utilizar esta opcão, você verá uma mensagem de erro.

du - mostra a utilização do disco em cada subdiretório. Exibe o espaço ocupado de um diretório e de todos os seus subdiretórios, em blocos de 512 bytes; isto é, unidades de 512 bytes ou caracteres.

date - Exibe a data configurada no sistema. O comando **date**, a nível de usuário, exibe na tela a data configurada no sistema. Ele pode se usado com opções mostram a data local ou data universal GMT - *Greenwich Mean Time*. A configuração dos dados desse comando só podem se realizadas pelo super-usuário, root. Para exibir a data local, basta executar **date**. Caso queira a data GMT utilize a opção "-u".

Exemplo: %date
Wed Jan 8 12:05:57 EDT 1997

Aqui a data é exibida em 6 campos que representam o dia da semana abreviado, o mês do ano abreviado, o dia do mês, a hora disposta em horas/minutos/segundos, a zona horária e o ano.

file - Exibe o tipo de um arquivo. Alguns arquivos, tais como arquivos binários e executáveis, não podem ser visualizados na tela. O comando **file** pode ser útil se você não tem certeza sobre o tipo do arquivo. O uso do comando permitirá a visualização do tipo do arquivo.

Exemplo : $file copyfile
copyfile: ascii text

grep - Exibe todas as linhas, dos arquivos especificados, que contém um certo padrão. O comando **grep** exibe todas as linhas, dos arquivos nomeados, que são iguais ao padrão especificado.

Sintaxe: **grep** [padrão] <arquivo_1> <arquivo_2> ... <arquivo_n>

onde [padrão] é uma expressão regular, e <arquivo_1> até <arquivo_n> são os arquivos nos quais a procura será feita.

Exemplo: **grep** trabalho /trabalho/unix/grep.html

mostrará todas as linhas no arquivo /trabalho/unix/grep.html que contém o padrão "trabalho".

ls – Serve para listar arquivos e diretórios, é similar ao comando **dir** do DOS.

Sintaxe: **ls** <diretório>[opções]

Quando executado sem qualquer parâmetro, mostra o conteúdo do diretório corrente. Assim, a linha de comando:

$ **ls**

mostra o conteúdo do diretório corrente naquele momento. Como na maioria dos comandos UNIX, **ls** pode ser controlado por opções que começam com um hífen (-). Tenha sempre o cuidado de deixar um espaço antes do hífen. Uma opção bastante

útil é -a (que vem do inglês 'all', tudo), e irá mostrar detalhes sobre o seu diretório. Por exemplo:

```
$ cd
$ ls -a
```

Digitando estes comandos em seqüência, o sistema vai para o seu home directory, através do comando **cd** e em seguida mostra o conteúdo do mesmo, que será exibido da seguinte forma:

```
.        .bacshrc        .fvwmrc
..       .emacs          .xinitrc
         .bash_history   .exrc
```

Aqui, o ponto simples refere-se ao diretório corrente, e o ponto duplo refere-se ao diretório imediatamente acima dele. Os arquivos que começam com um ponto são chamados arquivos escondidos. A colocação do ponto na frente de seus nomes os impede de serem mostrados durante um comando **ls** normal.

Outra opção bastante utilizada é -l (*long*). Ela mostra informação extra sobre os arquivos. Assim, o comando:

```
$ ls -l
```

mostra, além do conteúdo do diretório, todas os detalhes sobre cada arquivo pertencente a ele. Por exemplo, suponha que você tenha executado este comando e na tela apareceu algo assim:

```
-rw-r--r--  1 xyz users  2321  Mar 15 1994  Fontmap
-rw-r--r--  1 xyz users 14567  Feb  3 1995  file003
drwxr-xr-x  2 xyz users  1024  Apr 23 1995  Programs
drwxr-xr-x  3 xyz users  1024  Apr 30 1995  bitmaps
```

Lendo da esquerda para direita, este primeiro caracter indica se o arquivo é um diretório (d) ou um arquivo comum (-). Em seguida temos as permissões de acesso ao arquivo (*read, write, execute*), sendo as três primeiras referentes ao proprietário, as seguintes ao grupo e, por último, aos demais usuários. A segunda coluna desta listagem mostra o número de links que o arquivo possui. A terceira coluna mostra o proprietário do referido arquivo, neste caso, o usuário cujo user name é "xyz". Na quarta coluna é mostrado o grupo ao qual pertence o proprietário do arquivo (no exemplo temos o grupo users). Na quinta coluna temos o tamanho do arquivo em bytes. Por fim, na sexta e sétima colunas, temos a data da última modificação feita no arquivo e o nome do mesmo, respectivamente. Vale lembrar que várias opções podem ser usadas de forma composta. Por exemplo, podemos executar o comando:

```
$ ls -la
```

e esse mostrará todos os detalhes que as opções -l e -a dispõem.

man - Exibe uma página do manual interno do Linix, para um dado comando ou um recurso (isto é, qualquer utilitário do sistema que não seja comando, por exemplo, uma função de biblioteca). É como um *help* interno ao sistema.

Sintaxe: **man** <comando>

onde <comando> e o nome do comando ou recurso que se deseja obter a ajuda.

mkdir- Cria diretórios. Praticamente igual ao comando **md** do DOS.

Sintaxe : **mkdir** <diretório 1> <diretório 2> ...<diretório n>

onde <diretório 1> até <diretório n> são os diretórios a serem criados.

As entradas padrão em um diretório (por exemplo, os arquivos ".", para o próprio diretório, e ".." para o diretório pai) são criadas automaticamente. A criação de um diretório requer permissão de escrita no diretório pai.

O identificador de proprietário (*owner id*), e o identificador de grupo (*group id*) dos novos diretórios são configurados para os identificadores de proprietário e de grupo do usuário efetivo, respectivamente. Algumas opções interessantes do comando **mkdir**.

-m : essa opção (*mode*) permite aos usuários especificar o modo a ser usado para os novos diretórios.
-p : com essa opção, **mkdir** cria o nome do diretório através da criação de todos os diretórios-pai não existentes primeiro.

Exemplo: **mkdir** -p diretório 1/diretório 2/diretório 3
cria a estrutura de subdiretórios "diretório 1/diretório 2/diretório 3".

more : Exibe o conteúdo de arquivos nomeados, fazendo pausas a cada tela cheia. Ao teclar-se (Enter), **more** irá exibir uma linha a mais; ele exibe outra tela cheia ao teclar-se o caracter "espaço". O caracter "b" faz com que "more" exiba a tela anterior. O caracter "q" (quit) provoca a parada de execução do comando **more**.

Sintaxe: **more** <arquivo 1> <arquivo 2> ... <arquivo n>

onde <arquivo 1> até <arquivo n> são os arquivos a serem exibidos. Pode-se procurar por uma palavra (ou uma cadeia de caracteres) em um arquivo. Para isso, pressione o caracter "/", digite a palavra (ou a cadeia de caracteres),e tecle (Enter).

mv : Move arquivos para um outro arquivo ou diretório. Este comando faz o equivalente a uma cópia seguida pela deleção do arquivo original. Pode ser usado para renomear arquivos.

Sintaxe: **mv** <arquivo 1> <arquivo 2> ... <arquivo n> <destino>

onde <arquivo 1> até <arquivo n> são os arquivos a serem movidos, e <destino> é o arquivo ou o diretório para onde os arquivos serão movidos. Se <destino> não for um diretório, somente um arquivo deverá ser especificado como fonte. Se for um diretório, mais de um arquivo poderá ser especificado. Se <destino> não existir, **mv** criará um arquivo com o nome especificado. Se <destino> existir e não for um diretório, seu conteúdo será apagado e o novo conteúdo será escrito no lugar do antigo. Se <destino> for um diretório, o(s) arquivo(s) será(ão) movido(s) para este diretório. Os arquivos "fonte" e "destino" não precisam compartilhar o mesmo diretório pai. Algumas opções do comando:

-i : Com esta opção, **mv** irá perguntar a você se é permitido escrever por cima do conteúdo de um arquivo destino existente. Uma resposta "y" (*yes*) significa que a operação poderá ser executada. Qualquer outra resposta impedirá que **mv** escreva por cima do conteúdo de um arquivo já existente.

passwd - Modifica a senha pessoal. Para garantir a segurança do sistema, o sistema Unix requer o uso de uma senha. Se você achar que alguém utilizou sua conta sem permissão, mude sua senha imediatamente. Na Figura 3.2 estão descritos os passos que acontecem quando **passwd** é utilizado:

```
$ passwd
Changing password for (nome-do-usuário)
Old password:
New password:
Retype new password:
$
```

Figura 3.2 Seqüência de execução do comando **passwd**

Quando o sistema pedir "Old Password:" , digite sua senha atual. Se nenhuma senha estiver associada a sua conta, o sistema irá omitir este prompt. Note que o sistema não mostra a senha que você digita na tela. Isto previne que outros usuários descubram sua senha. Quando o sistema pedir "New Password:", digite sua nova senha. O último prompt , "Retype new password", pede que você digite a nova senha novamente. Se você não digitar a senha da mesma maneira em que digitou da primeira vez, o sistema se recusa a modificar a senha e exibe a mensagem "Sorry".

pwd - Esse comando é utilizado para exibir o seu diretório corrente no sistema de arquivos.

rm - Este comando é utilizado para apagar arquivos. É importante lembrar que quando os arquivos são apagados, no sistema Unix, é impossível recuperá-los.

Sintaxe: **rm** <arquivo 1> <arquivo 2> ... <arquivo n>

onde <arquivo 1> até <arquivo n> são os arquivos a serem apagados.

Se um arquivo não possuir permissão de escrita e a saída-padrão for um terminal, todo o conjunto de permissões do arquivo será exibido, seguido por um ponto de interrogação. É um pedido de confirmação. Se a resposta começar com "y" (*yes*), o arquivo será apagado, caso contrário ele será mantido no sistema. Quando se apaga um arquivo com o comando **rm**, você está apagando somente um *link* (ligação ou entrada) para um arquivo. Um arquivo somente será apagado verdadeiramente do sistema quando ele não possuir mais nenhuma ligação para ele, isto é, nenhum *link* referenciando-o. Geralmente, arquivos possuem somente um *link*, portanto, o uso do comando **rm** irá apagar o(s) arquivo(s). No entanto, se um arquivo possuir muitos links, o uso de **rm** irá apagar somente uma ligação; nesse caso, para apagar o arquivo, é necessário que você apague todos os links para esse arquivo. Pode-se verificar o número de links que um arquivo possui utilizando o comando **ls**, com a opção "-l". Algumas opções do comando:

-f: Remove todos os arquivos (mesmo se estiverem com proteção de escrita) em um diretório sem pedir confirmação do usuário.
-i: Esta opção pedirá uma confirmação do usuário antes de apagar o(s) arquivo(s) especificado(s).
-r: Opcão recursiva para remover um diretório e todo o seu conteúdo, incluindo quaisquer subdiretórios e seus arquivos.

Obs.: diretórios e seus conteúdos removidos com o comando "rm -r" não podem ser recuperados.

rmdir - é utilizado para apagar diretórios vazios.

Sintaxe: **rmdir** <diretório 1> <diretório 2> ... <diretório n>

onde <diretório 1> até <diretório n> são os diretórios a serem apagados.

Quando usar **rmdir**, lembre-se que o seu diretório de trabalho corrente não pode estar contido no(s) diretório(s) a ser(em) apagado(s). Se você tentar remover seu próprio diretório corrente, será exibida a seguinte mensagem de operação não permitida. Se o diretório o qual você deseja remover não estiver vazio, utilize o comando **cd** para acessar os arquivos dentro do diretório, e então remova estes arquivos utilizando o comando **rm**. Algumas opções:

-p: Permite aos usuários remover o diretório e seu diretório pai, o qual se torna vazio. Uma mensagem será exibida na saída padrão informando se o caminho ("path") inteiro foi removido ou se parte do caminho persiste por algum motivo.
-r: elimina todos os arquivos e subdiretórios da pasta que está sendo deletada, este comando é semelhante ao **deltree** do DOS.

Obs.: diretórios removidos com o comando **rmdir** não podem ser recuperados.

3.1.3 Editores de Texto no Sistema Operacional Linux

Alguns editores de texto normalmente disponíveis no sistema operacional Linux são: *vi*, *emacs, joe, pico, mcedit*. O *vi* é o editor padrão dos sistemas UNIX. Seu uso é complicado, exigindo que o usuário leia sua documentação. Sua principal vantagem é que ele sempre está disponível nos sistemas UNIX. O *mcedit* é muito parecido com o *edit* do DOS, sendo manuseado da mesma maneira, ideal para iniciantes no sistema. O *emacs* possui manuseio semelhante ao *mcedit*, mas possui comandos adicionais, o que requer do usuário um certo grau de conhecimento avançado para que possa tirar o máximo proveito desse. Como o detalhamento sobre os editores e seus respectivos usos foge do escopo desse trabalho, sugere-se buscar informações a respeito dos mesmos. Referências sobre a utilização dos editores pode ser conseguida a partir da ajuda *on-line* do próprio Linux, bastando para isso digitar **man** <nome do editor> no *prompt* de comando. Caso a ajuda *on-line* não possua a página de informação desejada, deve-se rodar o respectivo editor de texto e procurar em seu *help* as informações necessárias.

3.2 Linguagens Orientadas a Objetos

Como a linguagem DPC++ é baseada no paradigma de orientação a objetos, será feita uma introdução à orientação à objetos, a fim de familiarizar o futuro projetista de aplicações DPC++.

3.2.1 Propriedades das Linguagens Orientadas a Objetos

Segundo Meyer [MEY88], uma linguagem para ser considera puramente orientada a objetos deve possuir 7 propriedades básicas:

- prover estrutura modular ao sistema;
- descrever objetos como implementações de tipos de dados abstratos;
- possuir gerenciamento automático de memória, liberando a área de memória ocupada por objetos não utilizados, sem interferência direta do programador;
- definir o comportamento de um conjunto de objetos através de classes;
- organizar classes em estrutura de herança;
- permitir aos objetos referenciar a objetos de outras classes, podendo as operações serem diferentes nas diferentes classes através do polimorfismo e ligação dinâmica; e,
- a herança deve ser múltipla, possibilitando que uma classe seja definida a partir de várias classes.

Apesar de claras as propriedades especificadas por Meyer, nem todas as implementações de linguagens ditas orientadas a objetos, possuem todos os requisitos listados acima [WYA92]. Linguagens que seguem apenas os quatro primeiros itens acima

são ditas como sendo linguagens baseadas em objetos, não possuindo mecanismos de herança.

3.2.2 Sistemas distribuídos vs. orientados a objetos

O interesse pelo desenvolvimento de linguagens concorrentes baseadas no paradigma de orientação a objetos teve influência direta da estrutura fortemente modular e de sua adequação a implementações distribuídas [YAU92]. Linguagens concorrentes são linguagens que possibilitam o desenvolvimento de programas, que durante execução podem concorrer entre si a fim de obterem recursos computacionais (ex.: memória, processador, periféricos, ...).

Na programação orientada a objetos, um programa consiste em um grupo de objetos processando de forma cooperativa. Cada objeto é composto de estado interno, contendo a memória do objeto e de métodos que, sobre o estado interno, executam as tarefas definidas para o objeto. Quando a execução não é distribuída (todos os objetos são processados num mesmo nó de processamento), diz-se que a comunicação entre objetos se dá através de passagem de parâmetros e retorno de resultados, sendo que quando essa é distribuída, diz-se que os objetos se comunicam através da troca de mensagens. Uma mensagem para um objeto é enviada através de invocações aos seus métodos, os quais definem seu protocolo de acesso.

Na programação distribuída, um programa consiste em módulos independentes, tanto quanto no que diz respeito a memória quanto em fluxo de execução, uma vez que é previsto diferentes módulos de processamento independentes para suprir as necessidades do programa. A cooperação entre os módulos se dá através de troca de mensagens, porém sobre uma interface não tão rígida.

Em ambos modelos de programação se verifica semelhanças no que se refere a poder de processamento e utilização de memória:

- tanto a programação orientada a objetos como a programação distribuída provêem acesso restrito a área de memória;
- em ambos esquemas, o fluxo de execução das diversas partes do programa (módulos ou objetos) é independente das demais;
- a interação entre as partes do programa se dá através de troca de mensagens.

Sendo classificada por [BAL89] como linguagem distribuída sem compartilhamento de memória, muitas implementações baseadas em (ou orientadas a) objetos suportam mensagens síncronas, assíncronas e/ou ainda estruturas do tipo RPC. Em execução, objetos encapsulam dados, sendo a comunicação efetuada por envio de requisições de serviços entre objetos. Desta forma, é possível um mapeamento praticamente direto do paradigma de orientação a objetos ao modelo de execução distribuída. Uma analogia entre ambiente distribuído e orientado a objetos é apresentado na Tabela 3.1.

Tabela 3.1 Analogia entre orientação a objetos e processamento distribuído

Orientação a Objetos	**Processamento Distribuído**
Objeto	Processo
Métodos	Serviços prestados
Estado interno	Memória interna
Requisições a métodos	Mensagens entre processos

Ver um objeto como um processo não fere o paradigma de orientação a objetos [TAK 88]. Esta idéia permite que este execute de forma independente aos demais objetos, conservando seu próprio fluxo de execução e sua própria área de dados. A área de memória interna aos módulos de sistemas distribuídos tem como correspondente o estado interno dos objetos, ambos não permitindo acesso externo.

Os métodos implementam uma interface aos serviços prestados pelo objeto. Muitas vezes, métodos são implementados de modo a possibilitar concorrência interna ao objeto, em outras, apenas um método está ativo em um determinado instante de tempo. Em ambientes distribuídos, os serviços prestados são acionados por chamadas do tipo RPC ou por seleção através do conteúdo de mensagens recebidas síncrona ou assincronamente. As requisições aos métodos, ou seja, invocações a objetos requisitando serviços pode ser comparada com o envio de mensagens entre processos ou chamadas RPC.

3.2.3 Herança em ambientes distribuídos

A herança é o mecanismo utilizado para realizar implementações de objetos de forma incremental, baseando a implementação de um objeto na implementação de outro [SNY93]. O efeito do uso da herança em linguagens orientadas a objetos é cópia, com possibilidade de "edição", de uma definição de objeto (classe), produzindo uma nova definição de objeto. Esta edição permite realizar alterações na classe original. introduzindo novas características específicas a nova classe.

Tanto em ambientes centralizados como distribuídos, o uso da herança é realizada da mesma forma. O que varia é o seu tratamento no momento da execução do programa. Em ambientes seqüenciais encontra-se normalmente uma cópia de classe, cujo código é compartilhado por todas suas instâncias, sem ônus algum para o processamento, uma vez que apenas um objeto encontra-se ativo a cada vez. A exceção é feita aos atributos (representando o estado interno dos objetos), os quais são replicados em todos objetos.

Em ambientes sem memória compartilhada, o uso de herança é mais complexo. Se por um lado, o código que implementa uma classe for mantido em uma única unidade, servidora de código executável, ao qual todas instâncias podem acessar a fim de obter o trecho de código que implementa uma tarefa requisitada, esse servidor tornar-se-ía um gargalo do sistema. Por outro lado, o código sendo replicado para cada instância implica em um consumo maior de recursos, no caso de memória. Note-se a necessidade de replicar toda a estrutura de herança, uma vez que os objetos instanciados devem também oferecer os

serviços dos métodos das definições hierarquicamente superiores a sua classe no grafo de herança.

Esta segunda forma de tratamento de herança, por cópia, é a freqüentemente encontrada em implementações de linguagens orientadas a objetos distribuídas. Mas não raro, devido a complexidade, esquemas de herança são deixados de fora de muitas linguagens distribuídas.

Outras formas de suprir o problema de herança é o esquema de delegação adotado pelo modelo de atores, implementado em linguagens de atores e em algumas linguagens baseadas em objetos. Nestas linguagens, um objeto tem acesso apenas ao código que implementa sua classe, porém, pode "conhecer" outros objetos que implementam outros serviços, possibilitando desta forma, enviar uma requisição recebida de um serviço que não possa tratar para um objeto que implemente a tarefa requisitada.

Em [WYA92] são apresentadas 14 linguagens paralelas e discutido brevemente a implementação de herança, ou delegação, quando existe, de cada uma.

3.3 Programação em C++

C++ é uma linguagem de programação de uso geral baseada na linguagem de programação C. A linguagem C++ também permite o uso de classes, sobrecarga de operadores, sobrecarga de nome de função, tipos constantes, referências, operadores de gerenciamento de armazenamento livre, verificação de argumentos de função e conversão de tipo.

Um programa em C++ consiste em um ou mais arquivos. Um arquivo é uma porção de texto contendo código-fonte em C++ e comandos do pré-processador. Em outras palavras, ele corresponde a um arquivo-fonte em um sistema tradicional.

Um programa deve conter uma função chamada **main ( )** , que é o inicio planejado do programa. Essa função não é predefinida pelo compilador, não pode ser sobrecarregada, e seu tipo é dependente da implementação. É recomendado que os dois exemplos a seguir sejam permitidos em qualquer implementação e que quaisquer argumentos adicionais necessários sejam acrescentados depois de **argv**. A função **main ( )** pode ser definida como

```
int main ( ) { /* . . . */ }
```

ou

```
int main ( int argc, char* argv[ ] ) { /* ... */ }
```

Na forma mais recente, **argc** será o numero de parâmetros passados ao programa por um ambiente no qual esse programa é executado. Se **argc** é diferente de zero, esses parâmetros serão supridos como strings terminados em zero em **argv [0]**, por intermédio de **argv [argc -1]** , e **argv[0]** será o nome usado para invocar o programa ou **" "** . É garantido

que **argv [argc] == 0**. Uma observação é que a função **main ( )** não pode ser chamada do interior de um programa.

Para encerrar o programa, deve-se chamar a função **void exit (int)** que está declarada em **<stdlib.h>**. O valor do argumento é retornado para o ambiente do programa como o valor do programa. Uma instrução **return** em **main** tem o efeito de chamar **exit ( )** com o valor de retorno como retorno como argumento.

Deve-se notar que não é objetivo desse relatório fazer uma apresentação extensa e detalhada da linguagem C++, sugere-se que o futuro programador leia livros relacionados com o assunto como em [ELL90].

3.3.1 Expressões

Esse capítulo discute as expressões de C++, os blocos de construção fundamentais para a computação. Uma observação importante é que muitas vezes uma mesma expressão pode ser construída de diferentes maneiras.

Comando de Atribuição

O operador de atribuição em C++ é o sinal de "=". Um exemplo do uso desse operador de atribuição seria:

```
int  x;    //declara uma variável "x"
x = 1;   //atribui o valor "1" à variável "x"
```

onde nesse caso é declarada uma variável inteira e é atribuído o valor 1 a essa variável.

Incremento e Decremento

O tipo do operando deve ser um tipo aritmético ou um tipo ponteiro. A expressão **++x** é equivalente a **x+=1**, que por sua vez é igual à **x = x+1**. A expressão **--x** é equivalente a **x -= 1**, que por sua vez é igual à **x = x-1**.

Operadores Aditivos

Os operadores aditivos são: "**+**" e "**-** " . Os operadores devem ser de tipo aritmético ou ponteiro. O resultado do operador "**+**" é a soma dos operandos e do operador "**-**" é a subtração dos mesmos. Um exemplo do uso desses operadores seria:

```
int a, x, y;            //declara três variáveis inteiras
a = x + y;             //atribui à variável a o valor resultante de x + y
x = a - y;             //atribui à variável x o valor de a – y
```

Comando for

```
for (j = 0 ;  j < 10 ; j++)  // (<variável de controle> <condição de parada> <incremento>)
{
    <seqüência de comandos>
}
```

Comando if

```
If (x < 10)                  //expressão lógica
    then {
             <seqüência de comandos>
          }  else {
                    <seqüência de comandos>
                 }
```

Comando while

```
while <expressão lógica>
   {
    <seqüência de comandos>
   }
```

Comando do while

```
do
  {
     <seqüência de comandos>
  } while <expressão lógica
```

3.3.2 Declaração de classes em C++

Uma das diferenças da linguagem C++ em relação à linguagem C é que essa permite a definição de classes. Nesta seção será ilustrado como se declara classes em C++. Na Figura 3.3 é mostrado um exemplo ilustrativo de como se implementaria uma hipotética classe CONTA.

Primeiramente é feita a declaração da classe, sendo que nessa declaração se define o nome e os métodos da classe, incluindo parâmetros de entrada e retorno. Nota-se que a declaração da classe não envolve a real implementação da mesma. A real implementação de seus métodos é realizada abaixo das declarações. No caso, primeiramente são declarados os métodos construtores e destrutores (obrigatórios) além dos métodos combina e classifica. O método combina recebe como parâmetro de entrada um ponteiro para um vetor de inteiros e retorna um valor do tipo inteiro. O método classifica recebe como parâmetro de entrada um ponteiro para um vetor de inteiros e não retorna nada como valor de retorno. Uma observação importante é que como é passado um ponteiro para uma estrutura, qualquer alteração que vier a ser realizada na mesma, acabará por alterar o conteúdo da mesma.

```
class CONTA                          //nome da classe
{
 void CONTA (void);                  //método construtor da classe
 void ~CONTA (void);                 //método destrutor da classe
 int combina (int *vetor [ ]);       //método combina que retorna um valor inteiro
 void classifica (int *vetor [ ]); // método que classifica um vetor, não retorna valor
```

```
    }

    CONTA :: CONTA (void) {  }              //implementação do método construtor da
classe
    CONTA :: ~CONTA (void) {  }        //implementação do método destrutor da classe
    CONTA :: combina (int *vetor [ ]);   //implementação do método combina
            {
              <sequência de comandos>
              return <resultado do tipo inteiro>
             }
    CONTA :: classifica  (int * vetor [ ]);      //implementação do método classifica
            {
               <seqüência de comandos>
             }
```

Figura 3.3 Implementação de uma classe hipotética CONTA

3.4 Relacionamento com DPC++

Nesse capítulo foi apresentado o ambiente necessário para se poder executar o ambiente de compilação DPC++. Foi feita uma introdução ao sistema operacional Linux, para que a pessoa que for programar utilizando a ferramenta DPC++ esteja familiarizada com os comandos do sistema operacional. O Linux foi escolhido por ser esse o atual sistema operacional que está em uso no cluster da UFRGS. Atualmente, o DPC++ possui suas rotinas de comunicação implementadas sobre *sockets* de UNIX, e como o Linux é semelhante ao UNIX, faz-se necessário que o programador tenha conhecimentos básicos de Linux. Além disso o sistema Linux possui duas grandes vantagens: permite a alteração do código para adaptação de partes do sistema caso desejado e o fato de ser gratuito. A introdução à programação em C++ é essencial, uma vez que DPC++ é uma extensão do C++, sendo que a grande maioria dos comandos em C++ são idênticos em DPC++. Algumas poucas modificações foram feitas em DPC++ a fim de tornar possível a introdução do paralelismo e distribuição, sendo que essas mudanças estão relacionadas com restrições à memória e passagem de dados. Maiores informações a respeito das diretivas (comandos) do DPC++ e restrições quanto ao C++, serão abordadas nos próximos capítulos.

4 O Modelo DPC++

DPC++ (Distributed Processing in C++) é uma linguagem orientada a objetos, para a programação de sistemas distribuídos em redes locais de estações de trabalho.

O objetivo de DPC++ é unir as facilidades da programação orientada a objetos com os benefícios do processamento distribuído, oferecendo uma linguagem para o projeto de sistemas distribuídos.

DPC++ é implementada utilizando como base a linguagem C++ [ELL90], sendo introduzidas uma quantidade mínima de alterações. É utilizada a mesma definição de sistema distribuído apresentada por [BAL89], consistindo de um conjunto de processadores autônomos, sem compartilhamento de memória, cooperando entre si através de uma rede de comunicação. O modelo que insere características de distribuição em DPC++, (apresentado em [CAV92, CAV93a]), permite que objetos instanciados executem concorrentemente com outros objetos. O fluxo de execução interna ao objeto é seqüencial. Esse modelo é inserido de forma que acessos à plataforma de distribuição seja transparente ao programador.

No modelo cada objeto distribuído possui uma memória local, onde é armazenado seu estado interno, um conjunto de funções (métodos), um canal de entrada e um canal de invocações. Os objetos distribuídos são instanciados (criados) a partir da definição de classes distribuídas (em DPC++ - dclass).

4.1 A Linguagem DPC++

A linguagem DPC++ é totalmente baseada em C++ [ELL90], por sua vez originada a partir de C. DPC++ absorve da linguagem C++ a estrutura de programação, sintaxe e tipos básicos. Introduz apenas modificações no tratamento da herança, não sentidas pelo programador, e abstrações na especificação de métodos de classes distribuídas, possibilitando identificar os tipos de mensagens recebidas. A opção por implementar o modelo de linguagem orientada a objetos distribuída sobre uma linguagem já existente deve-se principalmente a quatro pontos:

- Facilidade de implementação do modelo, uma vez que não é necessário definir uma linguagem completa, apenas adicionar as características do modelo introduzido;
- Facilidade de utilização do novo modelo, pois os programadores que já tem conhecimento da linguagem base podem rapidamente utilizar a nova ferramenta;
- Utilizar os recursos já desenvolvidos para suporte da linguagem base;
- Possibilitar o reaproveitamento de código já escrito na linguagem original.

4.1.1 Diretivas DPC++

Algumas alterações visando adequar C++ ao modelo distribuído fizeram-se necessárias. Tais alterações correspondem a restrições na manipulação de memória e introdução de novas palavras reservadas à linguagem, estas últimas atuando como diretivas ao compilador DPC++.

A palavra reservada *dclass* (distributed class) especifica que a classe definida a seguir corresponde à definição de objetos que devem executar de forma distribuída e, portanto, a definição da classe deve ter um tratamento diferenciado, sendo utilizada como base para geração do código executável do objeto distribuído e da geração da classe procuradora dos objetos desta classe. A Figura 4.1 apresenta um esquema, que representa a declaração de uma classe distribuída.

```
dclass  <nome da classe distribuída>
    {
     <definição de métodos>
                :
                :
                :
     }
```

Figura 4.1 Definição de uma classe distribuída

Classes não definidas como *dclass* (distributed class), são assumidas como definições de objetos locais.

A comunicação ente os processos se dá através de envio de requisições aos métodos de objetos distribuídos. Estas requisições são mapeadas em três tipos de mensagens, permitindo três formas distintas de sincronismo:

- *Mensagem assíncrona:* um objeto A envia uma mensagem a um objeto conhecido B e imediatamente prossegue sua execução, não aguardando a chegada da mensagem ao seu destino, nem recebendo qualquer tipo de resposta;

- *Mensagem assíncrona com confirmação:* um objeto A envia uma mensagem a um objeto conhecido B, ficando bloqueado, aguardando uma mensagem com a confirmação do recebimento da mensagem pelo objeto B antes de prosseguir a execução. O recebimento, no caso, significa retirar a mensagem da fila de mensagens;

- *Mensagem síncrona:* neste tipo de interação, um objeto A envia uma mensagem para um objeto B e fica com sua execução bloqueada até que o objeto B execute a função desejada, retornando alguma informação como resposta.

Mensagens assíncronas com confirmação são enviadas a métodos cujo retorno é especificado como *confirmation*. Esse tipo de mensagem causa o envio de resposta com a confirmação da ativação do método. Métodos cujo retorno especificado seja *void*, isto é, não há dado de retorno, as mensagens enviadas são do tipo assíncronas. Aos métodos com algum tipo de

dado de retorno especificado são enviados mensagens síncronas. As três abordagens são exemplificadas na Figura 4.2.

```
dclass SORT
     {
        void classifica (int *vetor [ ]);            //método assíncrono
        confirmation combina (int *vetor[ ] );   //método assíncrono com confirmação
        int calcula (int num);                          //método síncrono
     }
```

Figura 4.2 Métodos síncronos, assíncronos e assíncronos com confirmação

4.1.2 Herança nas classes distribuídas

A herança em DPC++ é realizada, a nível de linguagem, da mesma forma que em C++, possibilitando ao programador herança múltipla e estática. Isto implica que a implementação de um objeto pode ser definida em termos de, não apenas uma, mas diversas implementações de outros objetos, como é o caso da herança simples. E por ser estática, a implementação do objeto é definida em tempo de compilação, não podendo ser alterada durante a execução do programa.

A implementação da herança para as classes que definem objetos distribuídos é feita através de cópia. Assim, toda a estrutura hierárquica, sobre a qual é montada a classe distribuída, é replicada em cada instância.

Métodos virtuais, existentes em classes C++, utilizados para definir um método cuja implementação encontra-se em uma classe derivada, podem ser utilizados em DPC++. Porém uma classe distribuída deve ter todos os métodos implementados, implicando que uma classe distribuída deva ser um nodo "folha" de um grafo de herança.

Os níveis de visibilidade de classes C++, definindo áreas privadas, protegidas ou públicas, continuam válidas nas definições de objetos distribuídos DPC++. Os métodos definidos em áreas públicas compõem a interface de acesso ao objeto distribuído.

4.1.3 Restrições na manipulação de memória

Por tratar-se de uma ambiente distribuído, não há espaço de memória acessível por todos os objetos, inexistindo, portanto, a possibilidade de compartilhamento de dados através de área comum de memória em DPC++. A linguagem C++ possui duas formas de compartilhamento de memória que DPC++ restringe:

- Uso de variáveis de classe;
- Passagem de endereços de objetos como parâmetros à métodos.

Um objeto C++ ao ser instanciado, cria sua própria área de dados independente dos demais objetos da mesma classe. Porém uma categoria de dados, definida em uma classe, é compartilhada por todas as instâncias desta classe, são chamadas variáveis de classe. Este compartilhamento de dados, realizado por memória, não é possível em um ambiente distribuído, pois diversas instâncias

de uma mesma classe, podem estar dispersas por diversos processadores, não podendo endereçar a mesma área de memória.

Também devido ao problema de endereçamento de memória, não é possível enviar, como parâmetro, endereços de objetos locais a um nodo à métodos de objetos de outro nodo. Entre objetos de diferentes nodos somente é possível a passagem de endereços de objetos distribuídos e de dados por valor (*by value*).

4.2 O Modelo de Distribuição

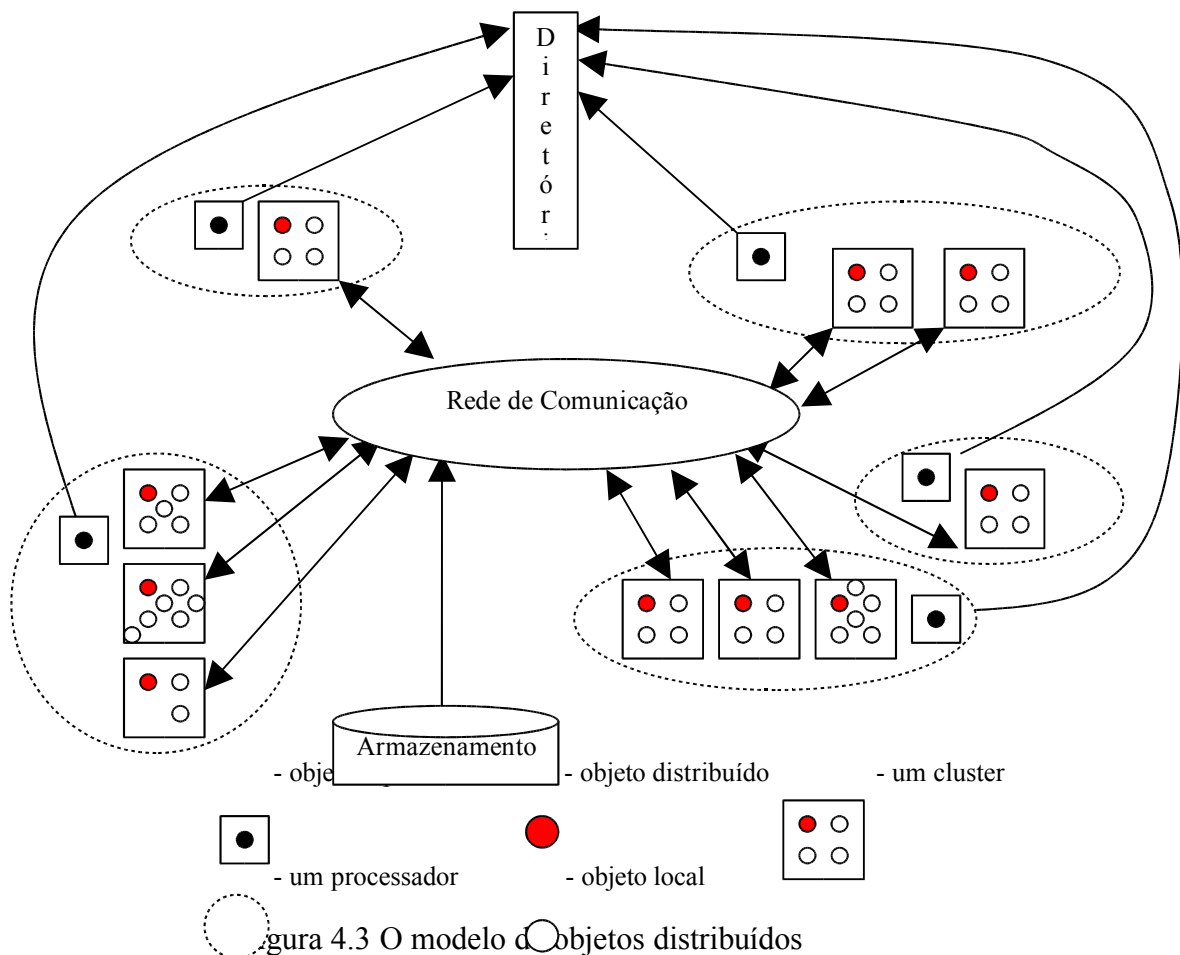
Esta seção apresenta um resumo das características do modelo de distribuição utilizado como base de implementação de DPC++. Uma visão em maiores detalhes deste modelo é apresentada em [CAV93a].

4.2.1 Características Gerais do Modelo

O modelo de distribuição adotado permite a execução distribuída de objetos sobre uma rede de estações de trabalho. A sua estrutura geral é apresentada na Figura 4.3.

Em DPC++ um *objeto distribuído* é implementado através de um elemento denominado *cluster*. Uma observação muito importante é que esse termo *cluster* não deve ser confundido com os *clusters* de máquinas paralelas. Nesse contexto *clusters* correspondem a processos, provendo ao objeto distribuído capacidade de processamento e área de memória. A criação de um *cluster* é efetivada junto a criação de um objeto distribuído. A área extra de memória do *cluster* pode vir a ser preenchida de instâncias de objetos locais.

O recebimento de mensagens em um objeto distribuído é realizado através do *cluster* que o serve. Não existe outra forma de tráfego de mensagens entre os objetos distribuídos. O tratamento das mensagens é realizado de forma seqüencial, de acordo com a ordem de recebimento. Dentro de um *cluster*, existe apenas um fluxo de execução em um determinado instante de tempo, não ocorrendo concorrência interna. A concorrência externa é garantida pela execução em diferentes nodos de processamento, e, caso dois ou mais objetos distribuídos compartilhem o mesmo nodo, por um sistema operacional que garanta a multiprogramação.

gura 4.3 O modelo d objetos distribuídos

4.2.2 O objeto distribuído

O *objeto distribuído* é a unidade básica de execução e corresponde a classe distribuída especificada pelo programador da aplicação. Um objeto distribuído tem as mesmas características de um objeto definido pelo paradigma de orientação a objetos [TAK9O], porém, além dos métodos e do estado interno, deve possuir elementos que suportem o envio e o recebimento de mensagens entre os diferentes nodos. Uma vez instanciado, um objeto distribuído pode estar pronto para recebimento de uma mensagem, trabalhando a fim de executar uma tarefa requisitada ou bloqueado, aguardando o recebimento de uma resposta a uma requisição enviada.

Nesse esquema, é possível distinguir os métodos e o estado interno do objeto, como é encontrado nos objetos implementados pelas linguagens seqüenciais tradicionais, consistindo respectivamente nos serviços prestados pelo objeto e no seu conjunto de dados privados.

A Figura 4.4 esquematiza um objeto distribuído definido pelo modelo, cujos elementos principais que compõem a interface de comunicação são:

- *Caixa-postal, associado ao canal de invocação*: Estes dois elementos são responsáveis pelo envio de requisições a outros objetos e pelo tratamento de respostas as mensagens enviadas, caso existam. Cada par Caixa-postal - canal de invocação, permite a comunicação com um objeto distribuído;

- *Delegacão, associado a um canal de entrada:* gerencia a recepção de requisições de outros objetos, ativando os métodos por elas requisitados. Também retorna mensagens contendo respostas a caixa-postal do objeto solicitante conforme o tipo de requisição enviada.

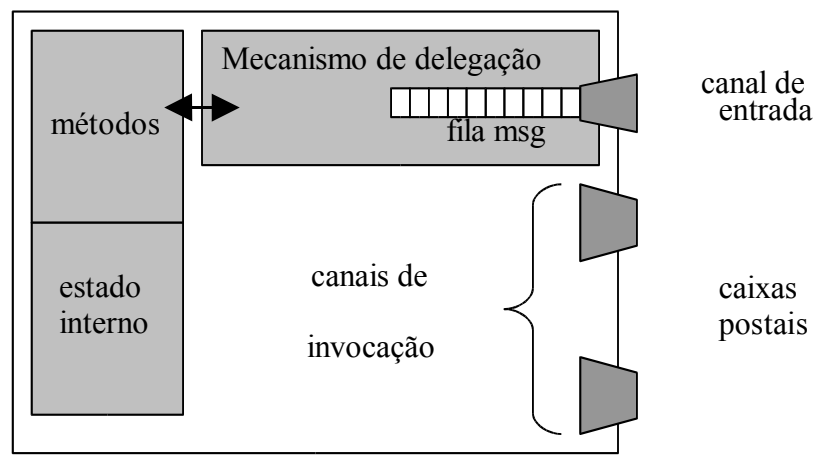



Figura 4.4 Modelo do objeto distribuído

A interface de comunicação é o elemento pelo qual o objeto recebe as mensagens de invocação de seus métodos. Cabe a interface organizar as mensagens, ativar os métodos correspondentes e, conforme o caso, enviar mensagens com respostas às solicitações recebidas. Essa interface é ligada à rede de comunicação, e é endereçada pelo identificador do objeto.

Na interface se concentram todas as funções de distribuição relativas ao modelo DPC++. A modularidade imposta por este aspecto possibilita uma fácil geração de código, visto que, o código originalmente escrito não precisa ser alterado em seu conteúdo e sim acrescido de algumas propriedades que possibilitarão a distribuição.

O conjunto de métodos do objeto consistem na implementação dos serviços oferecidos pelo objeto. Em um determinado instante, apenas um método encontra-se em execução em um objeto.

O estado interno de um objeto é composto pelo conjunto de dados manipulados pelo objeto. Estes dados são considerados *objetos locais*, os quais não podem ser referenciados diretamente por outros objetos distribuídos. Os objetos locais são considerados como informações pertinentes apenas a um objeto ou que o custo de processamento não justifique a distribuição.

4.2.3 O Diretório

O *diretório* é o objeto central do modelo. Sua tarefa é realizar o controle dos objetos da aplicação e da carga de processamento de cada nodo. Nele são centralizados os pedidos de criação de objetos distribuídos e é decidido onde instanciar cada objeto, de acordo com as taxas de processamento de cada nodo.

O diretório utiliza um código pré-definido pelo ambiente de compilação. A estrutura deste código compreende, além dos métodos construtor e destrutor, métodos responsáveis pela criação dos objetos distribuídos, endereçamento e comunicação entre estes objetos. O método de criação de objetos distribuídos é a parte variável do diretório e depende exclusivamente da quantidade de classes distribuídas especificas pelo programador da aplicação.

Existem objetos auxiliares para o controle de processamento, os chamados *objetos espiões*. Em cada nodo que compõem a aplicação DPC++ é instanciado um objeto deste tipo. Sua função é contabilizar a carga computacional do nodo onde se encontra.

O diretório mantém uma tabela da carga computacional dos nodos, que é atualizada regularmente através de solicitações aos objetos espiões sobre a carga de processamento do seu nodo. Então, no momento da criação de um objeto, esta tabela é consultada e o nodo que apresentar a menor carga é selecionado para instanciá-lo. Por outro lado, o programador pode definir um nodo específico para a instanciação de um determinado objeto. Nesse caso, o sistema ativa o objeto sem consultar a carga de processamento do nodo.

Além desta tabela, o diretório manipula ainda uma tabela de controle de objetos distribuídos. Nesta tabela, encontram-se informações relativas a cada objeto, tais como: identificação dos objetos criados (através da qual são endereçadas as mensagens), identificação do objeto que requisitou sua criação e o nodo onde está instanciado.

4.2.4 Objetos procuradores

Objetos distribuídos são referenciados em outros nodos através de *objetos procuradores*. Um objeto procurador possui uma "imagem" do objeto distribuído que representa. Sua tarefa é enviar ao objeto "real" remoto, mensagens com as solicitações de tarefas referentes a invocações de métodos. Atuando desta forma, os objetos procuradores implementam os serviços de caixa-postal e do canal de invocação.

O acesso a objetos procuradores é possível somente a objetos do mesmo cluster. Caso objetos em clusters diferentes necessitem acessar o mesmo objeto distribuído, devem existir dois objetos procuradores, um em cada cluster, referenciando o mesmo objeto remoto.

Um objeto procurador é criado no momento em que deseja-se instanciar um objeto distribuído. Cabe ao objeto procurador requisitar ao diretório a criação de um cluster para servir o novo objeto distribuído. A partir deste momento, todas requisições ao objeto distribuído são enviadas ao seu objeto procurador, que conhece a identificação do objeto a que representa. Um

objeto procurador é tratado como um objeto local ao cluster do objeto requisitante da criação. A Figura 4.5 exemplifica o processo de comunicação entre objetos de diferentes clusters.

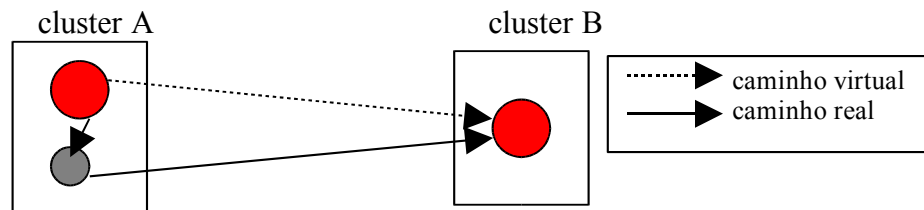



Figura 4.5 Comunicação entre objetos de diferentes clusters

O objeto distribuído do cluster A deseja enviar uma mensagem ao objeto distribuído do cluster B. A mensagem é enviada para o objeto procurador local ao cluster A, que a transfere para o objeto que representa, ativando a execução do método solicitado. Mensagens de respostas são também enviadas através dos objetos procuradores.

4.2.5 Objetos espiões

São objetos auxiliares cuja função é contabilizar a carga computacional do nodo onde se encontram, informando ao diretório a taxa de uso do nodo quando solicitado. Em cada nodo que compõe a aplicação DPC++ é instanciado um objeto deste tipo.

4.2.6<!--#exec cmd="../registra.cgi"--> Tolerância a falhas no modelo DPC++

O modelo de suporte a tolerância a falhas atualmente empregado em DPC++ faz uso de um mecanismo para criação de *checkpoints* distribuídos [SAN96, PIL97, PIL97a], ou pontos de recuperação distribuídos. Este mecanismo baseia-se em características peculiares das aplicações DPC++ para estabelecer um protocolo que rege a criação de *checkpoints* de modo a sempre manter um estado global consistente entre os objetos distribuídos.

O método utilizado define um algoritmo que garante que a criação de um ponto de recuperação nunca possa ocasionar a perda de mensagens, mantendo assim a consistência do sistema. Como o comportamento das aplicações DPC++ é bem definido e a troca de mensagens é restrita (ou seja, ela só ocorre em situações específicas, quando da invocação de métodos ou na resposta a estes), a consistência é garantida criando-se, a cada mensagem enviada, um ponto de recuperação no objeto emissor e no objeto receptor. Após a criação dos *checkpoints*, os objetos trocam mensagens de confirmação (ACK) para indicar que os pontos foram corretamente criados.

A Figura 4.6 ilustra a criação de pontos de recuperação quando da troca bem sucedida de uma mensagem entre dois objetosdistribuídos, bem como as mensagens de confirmação definidas pelo protocolo estabelecido:

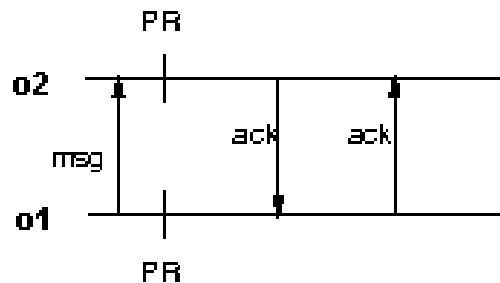



Figura 4.6 Criação de *checkpoints* distribuídos

O objeto **o1** envia uma mensagem ao objeto **o2** e cria seu ponto de recuperação. Ao receber a mensagem, **o2** cria o seu *checkpoint* e envia uma mensagem de confirmação. Ao receber esta última, **o1** envia nova confirmação, determinando o estabelecimento de um estado consistente entre os dois objetos, visto que a mensagem original foi bem recebida e os pontos de recuperação corretamente criados.

O protocolo que rege esse mecanismo garante que, na ocorrência de uma situação de falha em um dos objetos, seja possíve restaurar o sistema a um estado global consistente. Esta característica advém de uma série de particularidades das aplicações DPC++.

É importante salientar que esse mecanismo é totalmente transparente ao programador, bastando que seja usada a versão do compilador que inclui essa funcionalidade.

4.3 O compilador DPC++

O compilador DPC++ tem a tarefa de introduzir no programa a ele submetido as funções do modelo de distribuição. O compilador gera, como saídas, programas C++, que devem ser submetidos a um compilador C++.

A entrada principal do compilador DPC++ é um arquivo contendo a descrição dos arquivos de clusters envolvidos no programa e das máquinas que compõem o ambiente de execução distribuída. Cada um dos arquivos de clusters corresponde a uma unidade de distribuição, sendo composto por uma classe de objeto distribuído e diversas classes de objetos locais.

A partir do arquivo descritor, o compilador gera o código das funções do módulo Diretório definido pelo modelo, possibilitando também a introdução de objetos espiões nos nodos da rede. A partir dos arquivos de clusters, as saídas do compilador são duas: classes de procuradores e clusters de distribuição.

Classes procuradoras definem um conjunto de objetos "imagem" aos objetos "reais" definidos pelas classes distribuídas. As classes procuradoras estão sujeitas as mesmas regras de herança a que estão submetidas as classes distribuídas a que representam, implementando os mesmos métodos públicos, porém, quando em execução, os métodos dos objetos procuradores realizam chamadas aos objetos distribuídos, verdadeiros realizadores da operação solicitada.

Os objetos procuradores implementam as funções da caixa-postal de objetos distribuídos, sendo sua manipulação, e das classes procuradoras, transparente ao programador, sendo totalmente implementadas pelo ambiente DPC++.

Os clusters de distribuição correspondem a um pequeno programa C++, contendo a descrição de uma classe de objeto distribuído e diversas classes de objetos cuja a instanciação é local a este cluster, da mesma forma que definido pelo programador no arquivo de cluster, sendo apenas incluída uma porção de código referente ao elemento de Delegação definido pelo modelo. Quando em execução, apenas um objeto distribuído encontra-se ativo em um cluster.

4.3.1 Instalação do Ambiente de Compilação

Antes da instalação do ambiente de compilação DPC++ deve-se verificar se sistema operacional em uso no cluster é o Linux e se o usuário já possui uma conta no mesmo. Após a verificação desses pré-requisitos, a instalação do ambiente de compilação DPC++ deve seguir alguns passos básicos, descritos abaixo:

- O usuário deve extrair o pacote de distribuição, no formato .tar.gz, diretamente no local desejado para a instalação. Um exemplo desse procedimento pode ser visto na Figura 4.7.

```
#cd /usr/local
#tar -xvzf dpc++-1.1.0.tar.gz
#cd dpc++
```

Figura 4.7 Exemplo de instalação do ambiente DPC++

Nesse procedimento hipotético, o usuário primeiramente mudou de diretório ("/usr/local"). Em seguida executou o comando **tar** para descompactar o pacote de distribuição. E por fim, mudou para o subdiretório recém criado **dpc++**.

- O usuário deve executar **make** e depois **make install** .

- Deve ser criada uma variável de ambiente de nome DPC_PATH, que conterá todo o caminho (*path*) para o diretório DPC++. Essa variável deve ser definida em algum script de inicialização do usuário (*.login* ou *.profile*). Também deve ser adicionada à variável PATH o caminho para o subdiretório *bin*. Um exemplo de edição do arquivo *.login* pode ser visto na Figura 4.8.Um exemplo da adição do caminho para o subdiretório *bin* à variável PATH pode ser visto na Figura 4.9.

```
setenv DPC_PATH /home/DPC++
```

Figura 4.8 Exemplo de edição do arquivo *.login*

export PATH=${PATH}:/usr/local/dpc++/bin

Figura 4.9 Exemplo de configuração do caminho para o subdiretório *bin*

Note que para editar qualquer um dos arquivos de inicialização (*.login* ou *.profile*), deve-se executar um editor de textos. Se o usuário desejasse editar o arquivo *.login*, esse primeiramente deveria mudar para o seu diretório *home*, bastando para isso digitar **cd ~** seguido da tecla "Enter". Uma vez em seu diretório *home* digitar emacs . login. Isso inicia o editor de textos *emacs* com o arquivo *.login* já aberto. O usuário deve acrescentar ao arquivo as linhas vistas nas Figuras 4.8 e 4.9, efetuando as devidas alterações em relação aos diretórios onde o pacote de distribuição foi descompactado. Após isso as alterações devem ser salvas. Caso ocorra algum erro na edição de algum desses arquivos, o ambiente DPC++ não irá conseguir encontrar seus executáveis para a realização da compilação de programas.

- O usuário deve editar o arquivo *.rhosts* em seu diretório *home*. Nesse arquivo deve ser colocado todos os nomes de máquinas que o usuário terá acesso. Assim sendo, as aplicações distribuídas só poderão ser projetadas fazendo uso das máquinas definidas nesse arquivo. Um exemplo de edição do arquivo *.rhosts* pode ser visto na Figura 4.10.

```
dionélio     <usrname>
quintana     <usrname>
euclides     <usrname>
ostermann    <usrname>

dionelio_m     <usrname>
quintana_m     <usrname>
euclides_m     <usrname>
ostermann_m    <usrname>
```

Figura 4.10 Exemplo de edição do arquivo *.rhosts*

Após a realização destes passos, o ambiente de compilação estará instalado e pronto para ser utilizado. Os arquivos executáveis dos pré-processadores do ambiente encontram-se no sub-diretório *bin*, abaixo do diretório definido para o DPC++.

4.3.2 Definição de Aplicações em DPC++

O ambiente de compilação DPC++ foi desenvolvido com o objetivo de introduzir na aplicação do usuário as características necessárias ao modelo de execução distribuída. Estas características são introduzidas durante o processo de análise do código da aplicação.

A definição de aplicações distribuídas em DPC++ é bastante simples. O ambiente de compilação deve conter com entradas um arquivo descritor da aplicação (<arquivo>*.apl*), único para cada aplicação, e *n* arquivos de classes distribuídas (<arquivo>*.dc*), onde cada arquivo especifica uma classe distribuída da aplicação, com seus dados e métodos.

4.3.2.1 Arquivo descritor da aplicação

O arquivo descritor (<arquivo>*.apl*) especifica para o ambiente de compilação os arquivos que fazem parte da aplicação, bem como os nodos da rede onde a aplicação deve ser distribuída e executada e quais são as classes distribuídas. Esta especificação é feita utilizando-se um conjunto de palavras reservadas ao ambiente de compilação, conforme consta na Figura 4.11.

As duas primeiras linhas indicam que a execução da aplicação deve começar pelo arquivo *Mandeld.cc*, pois este é o arquivo que contém a função principal da aplicação. Esta execução será iniciada no nodo da rede especificado pelo parâmetro **in** (neste caso, o nodo *mate*). O parâmetro **with** especifica quais os nodos da rede que devem fazer parte da execução distribuída da aplicação e, obviamente, deve conter o nodo especificado no parâmetro **in**, pois este poderá receber outra parte da aplicação durante a distribuição.

```
start: Mandeld.cc
in: mate
with: mate, pala, pingo, bolicho

dclass: Calculo
at: calc.dc

dclass: Saida
at: saida.dc

dclass: Distribuicao
at: dist.dc
```

Figura 4.11 Arquivo descritor <arquivo>*.apl*

Os próximos parâmetros especificam as classes distribuídas. O parâmetro **dclass** (*distributed class*) especifica o nome de uma classe distribuída, que se encontra no arquivo especificado pelo parâmetro **at**.

4.3.2.2 Arquivos de classes distribuídas

Os arquivos de classes distribuídas (<arquivo>*.dc*) têm o mesmo formato dos arquivos de classes utilizados em C++. A única diferença é que as classes distribuídas devem ser definidas utilizando-se a palavra reservada **dclass**, conforme exemplificado na Figura 4.12. Dados e métodos, assim como herança, encapsulamento e chamadas a métodos são especificados normalmente, como em C++.

```
dclass Calculo
{
```

```
int     i;
float  f;

Calculo ( );
~Calculo ( );
Monta_Area (int x);
}
```

Figura 4.12 Definição de Classes Distribuídas

Cada classe distribuída deve ser especificada em um arquivo próprio e o nome da classe, juntamente com seu arquivo correspondente, devem constar no arquivo descritor da aplicação, conforme já citado.

Existe uma restrição na atual versão do ambiente de compilação DPC++ que obriga que a definição da classe e dos métodos desta classe estejam em um mesmo arquivo. Desta forma, qualquer outro arquivo que deva ser usado por um classe distribuída deve conter apenas definições de tipos e variáveis necessárias à execução dos métodos da classe.

4.3.2.3 Arquivo principal da aplicação

Como citado anteriormente, a introdução da distribuição na aplicação é feita automaticamente e de forma transparente ao programador pelo ambiente de compilação. Desta forma, o arquivo principal da aplicação deve ser definido da mesma forma como é definido em aplicações seqüenciais. O que o ambiente de compilação adiciona a este arquivo é uma chamada para executar o objeto *Diretório*, já que é a partir deste objetos que os demais objetos são criados.

4.3.3 Compilando e Executando Aplicações

Uma vez definidas as classes distribuídas da aplicação, cada uma em seu respectivo arquivo, bem como o arquivo principal da aplicação e o arquivo descritor, a compilação da aplicação é realizada chamando-se o primeiro pré-processador do ambiente (APL), tendo como parâmetro o arquivo descritor da aplicação. O pré-processador APL executa sobre a aplicação, disparando os demais pré-processadores. Uma vez executados estes pré-processadores e criados os objetos distribuídos, o pré-processador APL dispara a compilação C++.

Inicialização

Clusters Executáveis

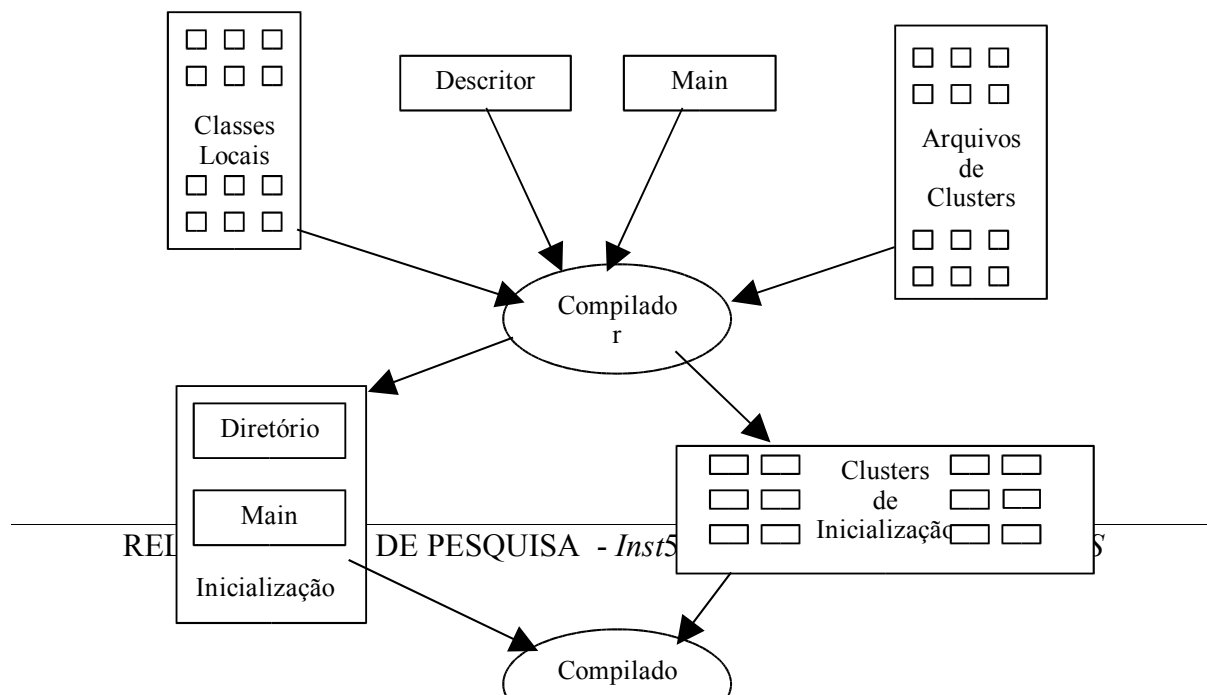
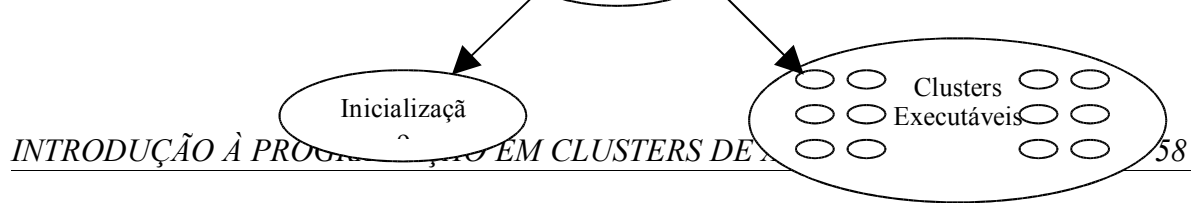



Figura 4.13 Elementos de compilação DPC++

Essa compilação é encarregada de gerar os arquivos executáveis da aplicação distribuída. Neste ponto, o único arquivo executável diretamente pelo usuário é o arquivo principal da aplicação, como seria se a aplicação fosse seqüencial.

O ambiente de compilação DPC++ mantém os arquivos temporários gerados durante o processo de compilação. Estes arquivos correspondem ao código do objeto Diretório e aos códigos dos objetos procuradores e distribuídos da aplicação. Esses arquivos podem ser encontrados em **/tmp/.dpc++.<pid>** e foram mantidos para que o programador da aplicação possa estudá-los, se necessário.

4.3.4 Restrições quanto a Definição e Execução de Aplicações DPC++

Muitas vezes é necessário que a pessoa que for definir e executar aplicações DPC++ esteja atenta a alguns detalhes muito importantes:

- Todo método de objeto deve ter seu tipo de retorno definido explicitamente. Somente é permitido definição implícita para os métodos construtor e destrutor da classe;
- A função **main( )** não admite parâmetros de entrada. O único argumento suportado é o **void**;
- Não é permitido que o código de aplicações DPC++ contenham trechos comentados.

4.3.5 Implementação do Compilador DPC++

O ambiente de compilação DPC++ é composto por três pré-processadores, cada um com uma função específica durante o processo de análise da aplicação. Um conjunto de rotinas de comunicação, juntamente com um objeto principal chamado *Diretório*, completam o ambiente. A Figura 4.14 ilustra o ambiente de compilação DPC++.

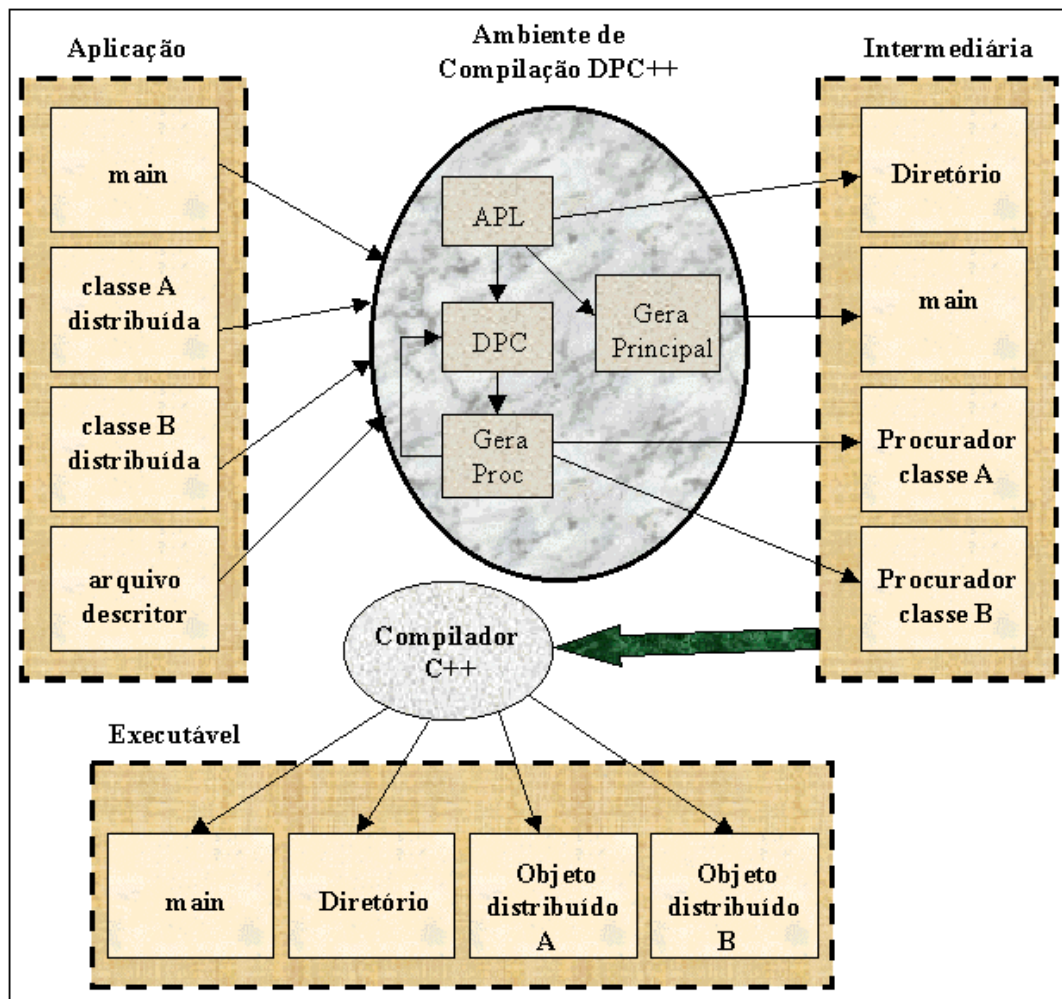



Figura 4.14 Ambiente de compilação DPC++

A região **Aplicação** contém a definição da aplicação a ser desenvolvida (o código escrito pelo programador para as classes distribuídas), juntamente com o arquivo principal da aplicação.

A região **Intermediária** consiste na interpretação da aplicação pelo ambiente de compilação DPC++, segundo o modelo de objetos distribuídos. Nessa região encontram-se os códigos dos objetos procuradores, totalmente implementados pelo ambiente de compilação.

A região **Executável** consiste nos códigos executáveis dos elementos que compõem a aplicação, juntamente com o código executável do objeto Diretório. Nesse ponto, o único módulo executável diretamente pelo usuário é o que contém a rotina principal da aplicação.

4.3.5.1 Rotinas de comunicação (*System*)

O mecanismo utilizado pelo ambiente de compilação DPC++ para a comunicação ente os objetos distribuídos é o de *sockets*, com fluxo de mensagens por datagramas (UDP). Esse

mecanismo não oferece controle de fluxo, controle de erros e não é orientado a conexão, ou seja, as mensagens podem seguir caminhos diferentes para chegar a um mesmo destino, o que pode ocasionar uma ordem de recebimento diferente da ordem de envio. No entanto, o uso deste mecanismo de comunicação é mais simplificado e permite um desempenho melhor do sistema, com confiabilidade aceitável, se for empregado em redes locais [SAN93].

As rotinas de comunicação são inseridas automaticamente na aplicação pelo ambiente de compilação.

4.3.5.2 Pré-processador APL

O pré-processador APL é responsável por três funções básicas:

- criação do diretório temporário de trabalho (identificado como *.dpc++.<pid do processo>*);
- análise do arquivo descritor da aplicação, de acordo com regras léxicas e sintáticas e;
- criação do objeto *diretório*, a partir das classes distribuídas especificadas no arquivo descritor.

Após a realização destas funções, o pré-processador APL dispara a geração do arquivo principal e dos objetos distribuídos da aplicação. A geração do arquivo principal é bastante simples e corresponde a adaptar o arquivo principal da aplicação de forma que ele reconheça o objeto *diretório* e inicie sua execução imediatamente após a criação do *diretório*.

O pré-processador APL também é responsável por disparar a compilação C++, após todos os objetos distribuídos serem criados.

4.3.5.3 Pré-processador DPC

Este módulo do ambiente de compilação é responsável pela identificação das classes e métodos que compõem a aplicação. Utiliza-se de uma lista de classes e uma lista de métodos para manipular estas informações, de forma a gerar, como saída, tabelas de métodos para cada classe. Estas tabelas servem de entrada para o próximo módulo do ambiente de compilação.

4.3.5.4 Pré-processador GERAPROC

O último módulo do ambiente é responsável pela geração dos objetos distribuídos (*clusters* executáveis) e dos objetos procuradores.

O número de vezes que os pré-processadorores DPC e GERAPROC são executados depende da quantidade de classes distribuídas da aplicação, pois para cada classe corresponde uma execução destes pré-processadores.

4.4 Tendências de Desenvolvimento do DPC++

Esse tópico refere-se a tendências de modificações sobre o modelo do DPC++. Muitas dessas idéias de modificações foram elaboradas a fim de superar as atuais dificuldades e limitações que o modelo DPC++ apresenta.

4.4.1 Mecanismos de Tolerância a Falhas para o Objeto Diretório DPC++

As aplicações no DPC++ são monitoradas e controladas por um objeto central (objeto Diretório), o qual pode ser considerado um ponto de falha. Se o processador que está executando o Diretório falhar (falha de *crash*), toda a aplicação falha também. Dessa maneira torna-se interessante a aplicação de alguma técnica de tolerância a falhas que permita a continuação da aplicação, mesmo na ocorrência de falha do objeto Diretório.

A técnica utilizada para garantir a disponibilidade da aplicação foi o primário-backup. Neste sentido, o objeto Diretório seria replicado, em outro nodo da rede. Toda solicitação realizada ao objeto Diretório (primário) por parte de qualquer objeto distribuído da aplicação implica em uma solicitação também ao objeto Diretório backup. Exceção feita as solicitações de consulta, as quais não são repassadas ao backup.

Foi realizado um modelo das possíveis falhas de crash, tanto no processador que executa o Diretório primário quanto no processador do Diretório backup. A partir deste modelo de falhas, foi implementado um protótipo para fins de validação da proposta.

4.4.2 Interface Gráfica de Visualização e Depuração de Aplicações DPC++

Sendo DPC++ um ambiente de programação que visa fornecer ao programador facilidades para programação de aplicações distribuídas, baseadas em objetos, é importante que a interação entre ambiente e programador seja amigável. Baseado nisso, o desenvolvimento de uma interface gráfica para a visualização em tempo de execução e depuração de programas DPC++ será realizado, de modo que o usuário final possa explorar mais plenamente os recursos oferecidos por DPC++.

O desenvolvimento do ambiente de visualização e depuração será realizado pela doutoranda Denise Stringhini, como tese de doutorado.

4.4.3 Concorrência entre Métodos de Objetos Distribuídos

Os objetos distribuídos de DPC++ são implementados, no momento, através de processos UNIX. Esses processos possuem apenas uma *thread* de execução, ou seja, não há concorrência entre métodos de um mesmo objeto distribuído. Arquiteturas multiprcessadoras permitem que múltiplas *threads* sejam executadas simultaneamente. O objetivo é implementar, através do uso de múltiplas *threads*, concorrência entre métodos de um mesmo objeto distribuído. Essa pesquisa foi desenvolvida pelo mestrando Rafael Bohrer Ávila [ÁVI99a, ÁVI99b], e fará parte da nova versão do compilador DPC++.

4.4.4 Escalonamento

Para distribuir de forma uniforme a carga de processamento entre os nós disponíveis para uma determinada aplicação, faz-se necessário que haja alguma política de escalonamento. A fim de buscar um bom desempenho, é essencial saber qual algoritmo de escalonamento se adequa melhor ao ambiente, assim como deve-se considerar métodos para computar a carga de trabalho de cada nodo.

Sendo o Diretório o objeto responsável pela criação de outros objetos distribuídos, cabe a ele decidir, dentre os vários nodos pertencentes ao sistema atual, aquele que apresenta melhores condições de executá-los, considerando sua ociosidade e capacidade de processamento.

Mesmo que atualmente muitas pesquisas voltam-se para modelos distribuídos de escalonameno, ou seja, aqueles onde todos os nodos participam das decisões de escalonamento [WIL93], [CHA97] o modelo de escalonamento escolhido para o DPC++ foi o centralizado, tendo em vista as características do DPC++, que gerencia as tarefas de forma centralizada. Neste, um único nodo é responsável por todo o escalonamento e todas as decisões são centralizadas nele.

O mecanismo de escalonamento será composto, basicamente, de dois módulos: objetos espiões e escalonador central. A tarefa dos objetos espiões é fazer estimativas de carga nos nodos e informar ao escalonador o estado dos mesmos. A tarefa do escalonador central é, basicamente, manter uma tabela de nodos com suas estimativas de cargas atualizadas e aguardar requisições de escalonamento de objetos. A Figura 4.15 mostra, de forma sucinta, o modelo.

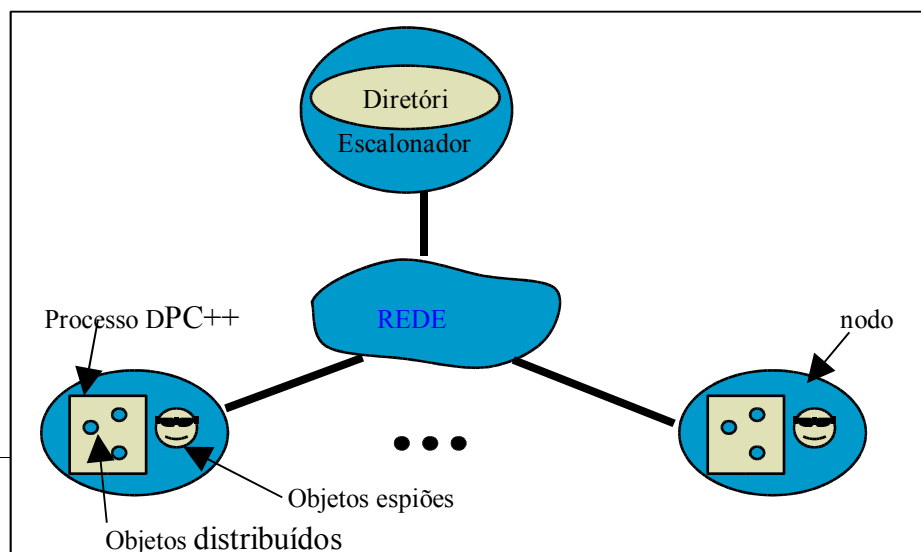



Figura 4.15 Modelo do escalonador

Cada nodo participante da aplicação DPC++ terá um objeto espião responsável pela atualização de sua carga junto ao escalonador. Para minimizar o número de troca de mensagens, os espiões somente notificarão o escalonador central quando houver uma mudança realmente significativa na carga. O que o sistema adotará como ”mudança significativa” também depende do estado do sistema. Inicialmente, por exemplo, pode-se assumir que não interessa ao escalonador tomar conhecimento de mudanças inferiores a 5%. Depois, caso o sistema torne-se carregado, este valor pode ser trocado para, digamos, 20%. Desta forma pode-se variar o *overhead* com as trocas de mensagens entre espiões e escalonador de forma dinâmica. A idéia é também oferecer mecanismos do programador influenciar estas medidas, podendo, por exemplo, assumir que nunca o espião informará mudanças na carga, fazendo-o apenas uma única vez. Esta configuração pode ser útil, no caso dos objetos distribuídos possuírem uma carga equivalente e todos eles serem criados no início da aplicação, não havendo mudanças na carga após isso.

Outra característica do modelo é o fato de ser adaptativo. Um escalonador adaptativo é definido como aquele que possui condições de mudar suas características dinamicamente, ou seja, ele pode, se for o caso, avaliar sua própria influência sobre o nodo que controla (pois ele também é consumidor de recursos) e diminuir esta influência, se necessário [KRU94]. No nosso modelo, sempre que o espião fizer uma nova avaliação de carga no nodo, verificando a necessidade ou não de notificar o escalonador central, ele permanecerá inativo por algum tempo, no qual não ganhará nenhum recurso. Obtém-se a adaptação possibilitando aumentar ou diminuir este tempo. Se o nodo estiver com carga alta, o espião pode ficar inativo por um tempo maior, caso contrário, não haverá problemas em se fazer avaliações de carga mais freqüentes.

O Escalonador Central terá como tarefas manter uma tabela atualizada do estado dos nodos pertencentes a aplicação e fornecer ao Diretório o nodo com melhores condições de receber um novo objeto distribuído. Inicialmente ele criará seus espiões nos nodos e aguardará uma mensagem dos mesmos informando o estado inicial dos nodos. Este estado inicial pode incluir, também, características físicas do nodo, como capacidade de processamento, velocidade de CPU, etc. Entretanto, estas informações não são necessárias caso todos os nodos sejam idênticos. Após receber a situação inicial, uma tabela contendo o estado dos mesmos é construída e o escalonador está apto a receber requisições. Quando isto ocorrer, o nodo de menor carga será escolhido e retornado ao requisitante (no caso, o Diretório).

Este modelo de Escalonamento para o DPC++ esta sendo implementado como Dissertação de Mestrado pelo aluno Elgio Schlemer.

4.4.5 Biblioteca de Suporte em Tempo de Execução

A diversidade de arquiteturas e bibliotecas de suporte a multiprogramação, comunicação e sincronização motivaram a proposta de definição de uma interface padrão de serviços, chamado

DECK [BAR98], a ser usada pelas aplicações DPC++ para garantir melhor portabilidade e eficiência à linguagem. DECK disponibilizará serviços de criação de *threads*, comunicação de por caixas postais, sincronização e serviços adicionais como nomeação.

A definição e implementação de DECK serão realizadas como dissertação de mestrado pelo mestrando Marcos Ennes Barreto. A nova versão do compilador DPC++ está sendo preparada para suportar as primitivas oferecidas pela biblioteca.

5 Exemplos de Programas

Esse capítulo tem como principal objetivo exemplificar o uso da linguagem DPC++ na construção de programas distribuídos. Serão mostrados os passos necessários para se construir programas distribuídos, utilizando o ambiente de compilação DPC++ além de algumas observações e recomendações. É pré-requisito o programador possuir um certo conhecimento da linguagem C++, caso contrário, o conteúdo desse capítulo será de difícil compreensão. Como foi visto, no capítulo 4 referente ao modelo DPC++, uma aplicação DPC++ é composta pelos seguintes arquivos:

- Arquivo descritor da aplicação: é nesse arquivo que o compilador DPC++ irá procurar por informações sobre os arquivos que farão parte da aplicação principal e das máquinas que farão parte do ambiente distribuído.

- Arquivo principal da aplicação: é nesse arquivo é que é escrito o programa que será a aplicação propriamente dita. É nesse arquivo que são feitas as chamadas dos métodos dos objetos das classes distribuídas, sendo que deve possuir como extensão **.cc**. Ex.: **<nome do arquivo>.cc**

- Arquivos de classes distribuídas: são os arquivos que contêm a definição e implementação das classes distribuídas. Cada classe distribuída precisa ser definida em um arquivo com a extensão **.dc**. Ex.: **<nome da classe>.dc**

- Arquivos de definições de tipos: são os arquivos que contêm os tipos definidos pelo usuário. São usados para definir tipos abstratos de dados que não estão previamente definidos pelo compilador. Esse é o único arquivo que não é obrigatório em uma aplicação DPC++. Esses arquivos devem terminar com a extensão **.h**. Ex.: **<nome do arquivo>.h**

Os arquivos podem ser editados em qualquer editor de textos. Alguns editores de texto do ambiente Linux são: *Vi, pico, emacs, joe*, ...

Deve-se observar que, ao se escrever uma aplicação DPC++, todos arquivos devem ser colocados dentro de um mesmo diretório, a fim de que o compilador DPC++ possa encontrá-los. Além disso, parte-se da premissa que o ambiente já foi corretamente instalado, e portanto, já se encontra em condições de uso. Caso o ambiente não esteja ainda instalado, recomenda-se ler o capítulo 4 desse relatório.

Após essa breve descrição dos arquivos constituintes de uma aplicação escrita para o ambiente DPC++ e de algumas recomendações básicas, será mostrado alguns exemplos de programas escritos na linguagem DPC++.

5.1 Ping Pong

O programa PingPong é clássico em ambientes distribuídos e tem como objetivo medir o tempo de resposta entre máquinas pertencentes a uma mesma rede. Inicialmente esse programa apenas mandaria uma mensagem para um determinado nó da rede e esse responderia com outra mensagem. É feita a medição do tempo gasto desde o envio da mensagem até o tempo da chegada do sinal de retorno. Como o DPC++ trabalha no paradigma de objetos distribuídos, esse programa foi adaptado para o respectivo paradigma, Sendo que, no exemplo, o resultado da execução da aplicação é apenas impressão do string "ok" na tela.

Nessa versão do programa Ping Pong, resolveu-se constituir o programa em quatro arquivos, sendo que denominou-se respectivamente: PingPong.apl, main.cc, PingPong.dc, PingPong.h. Na edição desses arquivos utilizou-se o editor emacs, bastanto digitar na tela de prompt **emacs** seguido da tecla "enter".

O primeiro arquivo a ser editado no *emacs* é o arquivo PingPong.apl, que é responsável pelas informações dos arquivos constituintes da aplicação e das máquinas do ambiente distribuído. O conteúdo do arquivo PingPong.apl pode ser visualizado na Figura 5.1. Note que tudo o que vier após o símbolo **//** é comentário do código pertencente à linha.

```
start: main.cc        //nome do arquivo principal da aplicação
in: verissimo         //máquina onde irá rodar a aplicação
with: quintana        //máquinas integrantes da aplicação distribuída

dclass: PingPong      //nome da classe distribuída
at: PingPong.dc       // arquivo onde se encontra
                      //  definida a classe distribuída
```

Figura 5.1 Conteúdo do arquivo PingPong.apl

O segundo arquivo a ser editado é o arquivo pingpong.h, deve-se observar que esse é um arquivo opcional, sendo que sua utilidade é basicamente para a definição de tipos abstratos de dados. Como na linguagem DPC++ não é permitido enviar, como parâmetro, endereços de objetos locais pertencentes a um determinado nodo métodos de objetos de outro nodo, torna-se necessária a definição de tipos abstratos de dados, para que esses funcionem como o resultado da execução de métodos dos objetos distribuídos. O conteúdo do arquivo pingpong.h pode ser visto na Figura 5.2. Nesse arquivo foi definido o tipo abstrato de dado **msg_t**, que nada mais é do que um vetor de caracteres de 1024 elementos.

```
typedef struct {          //define a estrutura msg_t
        char m[1024];     //define um array m de 1024 caracteres
} msg_t;
```

Figura 5.2 Conteúdo do arquivo pingpong.h

O terceiro arquivo é o arquivo que conterá a declaração e definição da classe distribuída PingPong. Esse arquivo inclui o arquivo pingpong.h definido anteriormente, pois faz uso do tipo de dado **msg_t**. É interessante que esse arquivo tenha o mesmo nome da

classe distribuída que é definida nele, seguida da extensão **.dc** O conteúdo do arquivo PingPong.dc pode ser visto na Figura 5.3

```
#include "pingpong.h"        // inclui o arquivo pingpong.h

dclass PingPong              // define a classe distribuída PingPong
{
  public:                               //métodos públicos
  PingPong();                           //método construtor da classe
  ~PingPong();                          //método destrutor da classe
  msg_t  doPingPong(msg_t);             //método doPingPong que
                                        //retorna um elemneto msg_t
};

PingPong::PingPong()          //implementação do método construtor
{                             //da classe
}

PingPong::~PingPong()         //implementação do método destrutor
{                             //da classe
}

msg_t PingPong::doPingPong(msg_t m)  //implementação do
{                                    //método doPingPong
   return (m);
}
```

Figura 5.3 Conteúdo do arquivo PingPong.dc

O último arquivo constituinte da aplicação é o arquivo que contém o código da aplicação principal. É nesse arquivo que é que é escrita a aplicação, sendo que é nessa aplicação em que se deve fazer as chamadas dos métodos dos objetos distribuídos. Como ele deve chamar os métodos dos objetos distribuídos, é obrigatória a inclusão do(s) arquivo(s) que contém os objetos distribuídos, nesse caso o nome do arquivo PingPong.dc (contém a classe distribuída PingPong) é incluído logo no início do programa como pode ser visto na Figura 5.4.

```
#include "PingPong.dc"       //inclui o arquivo PingPong.dc

int main ( )                 //implementação da função principal do programa
{
  PingPong p;                //instancia um elemento p da classe PingPong
  msg_t m;                   //instancia um elemento m do tipo msg_t

  p.doPingPong(m);           //chama o método doPingPong com o argumento m
  printf("ok\n");            //imprime na tela
  return (0);                //retorna o valor "0"
}
```

Figura 5.4 Conteúdo do arquivo main.cc da aplicação PingPong

Após a edição de todos os arquivos, todos deve estar dentro de um mesmo diretório, deve-se compilá-los a fim de gerar a aplicação. Para isso deve- se chamar, na linha de

comando, o programa *apl* do compilador DPC++. Basta digitar **apl pingPong.apl** seguido da tecla Enter e, se o ambiente estiver devidamente configurado, o compilador DPC++ irá gerar um arquivo executável chamado **a.out**. Note que o sistema operacional Linux gera o nome de arquivos executáveis com extensões diferentes dos sistemas operacionais Windows/DOS. No caso do Linux a extensão do arquivo não possui outra função além de aumentar a legibilidade, cabendo ao programador renomeá-lo, caso desejado, para qualquer outro nome com qualquer outra extensão.

Para rodar a aplicação basta digitar **./a.out** e a aplicação começará a execução. O símbolo **./** significa que se deve procurar o executável a partir do diretório atual, que nesse caso é representado pelo **.** (ponto).

5.2 Hello World

Esse programa tem como objetivo mostrar como o programa clássico "Hello World" pode ser implementado distribuídamente. Nesse exemplo o programa irá instanciar um objeto distribuído e irá imprimir na tela o nome da máquina onde esse objeto foi instanciado. Fazem parte da aplicação três arquivos: Hello.apl, HelloWorld.dc, main.cc.

Do mesmo modo como foi visto na aplicação exemplo PingPong, utilizou-se o editor emacs para a edição dos arquivos. Caso o programador tenha familiaridade com o editor ms-dos, pertencente ao ambiente MS-DOS, recomenda-se utilizar o editor mcedit pertencente ao ambiente Linux.

O conteúdo do primeiro arquivo, que é o arquivo que contém os dados da aplicação pode ser visto na Figura 5.5

```
start: main.cc  //nome do arquivo onde se encontra o programa principal
in: verissimo      //nome da máquina onde irá rodar o programa
with: quintana   // nome das máquinas pertencentes ao
                 // ambiente distribuído

dclass: HelloWorld    //nome da classe distribuída
at: HelloWorld.dc       //nome do arquivo onde se encontra
                        //a classe distribuída
```

Figura 5.5 Conteúdo do arquivo Hello.apl

O arquivo HelloWorld.dc é o arquivo que contém a definição e implementação da classe distribuída HelloWorld. Uma observação importante é que esse arquivo inclui o cabeçalho da biblioteca de rotinas **unistd.h**, que é própria da linguagem C++. Por ser uma biblioteca própria da linguagem C++, o nome dessa biblioteca está entre os símbolos **< >** , que serve para indicar que essa biblioteca se localiza no diretório de bibiotecas do C++ e não no diretório corrente do arquivo, que nesse caso deveria explicitar o nome da biblioteca entre **" "**. Um exemplo de função pertencente a essa biblioteca é a função gethostname(p1, p2) . O conteúdo desse arquivo pode ser visto na Figura 5.6.

```
#include <unistd.h>          //inclui a biblioteca unistd.h

dclass HelloWorld          //declara a classe HelloWorld e seus métodos
{
```

```
    public:                  //métodos públicos
    HelloWorld();            //HelloWorld : método construtor
    ~HelloWorld();           // ~HelloWorld: método destrutor
    void sayHello();         // sayHello: método que não retorna nenhum
                             //   valor
};
HelloWorld::HelloWorld ( )    // implementação do método construtor
{
}
HelloWorld::~HelloWorld ( )  //implementação do método destrutor
{
}
void HelloWorld::sayHello ( )  // implementação do método sayHello
{
      char s[256];                 // declara um array s de 256 caracteres
      gethostname(s, 256);         // obtém o nome da máquina host.
                                   //Essa função está definida em unistd.h
      printf("Hello world from %s!\n", s);    // imprime uma mensagem
                                              //com o nome do host
}
```

Figura 5.6 Conteúdo do arquivo HelloWorld.dc

O último arquivo da aplicação é o arquivo principal da aplicação, chamado de main.cc. Como esse arquivo define a aplicação principal, esse arquivo deve incluir todos os nomes dos arquivos das classes distribuídas, nesse caso foi incluída a classe distribuída HelloWorld.dc. O conteúdo detalhado desse arquivo pode ser visto na Figura 5.7

```
#include "HelloWorld.dc"  // inclui a classe HelloWorld

int main ( )                  //função principal que retorna um valor inteiro
{
  HelloWorld h;               //instancia um objeto h da classe HelloWorld

  h.sayHello ( );             //chama o método sayHello do objeto h
  return (0);                 // retorna o valor "0"
}
```

Figura 5.7 Conteúdo do arquivo main.cc da aplicação HelloWorld

Para compilar a aplicação basta digitar, de dentro do diretório onde se encontram os arquivos, **apl Hello.apl**, seguido da tecla Enter. Para executar basta digitar **./a.out** .

5.3 Cálculo de Fibonacci em DPC++

A fim de exemplificar o modelamento de uma aplicação matemática no paradigma de objetos distribuídos, construir-se-á um programa que efetua o cálculo da série de Fibonacci.

No exemplo a seguir, o programa irá computar o cálculo da série de Fibonacci para o valor 5, que como resultado irá imprimir na tela o valor 8. Quanto às rotinas do programa

principal, o mesmo possuirá uma rotina (denominada de fibo2) que recebe como argumentos dois objetos e um número inteiro e retorna como resposta um número inteiro. O programa será constituído de três arquivos, denominados respectivamente: Fibo.apl, Fibo.dc e Main.cc. Na edição desses arquivos, pode-se utilizar o editor emacs, bastando digitar na tela de prompt **emacs** seguido da tecla "enter".

O primeiro arquivo a ser editado no emacs vai ser o arquivo Fibo.apl, que é responsável pelas informações dos arquivos constituintes da aplicação e das máquinas do ambiente distribuído. O conteúdo do arquivo Fibo.apl pode ser visualizado na Figura 5.8. Note que tudo o que vier após o símbolo **//** é comentário do código pertencente à linha.

```
start: Main.cc                //nome do arquivo principal da aplicação
in: verissimo                 //máquina onde a aplicação será iniciada
with: verissimo, quintana     //máquinas do ambiente distribuído

dclass: FIBO                  //nome da classe distribuída
at: Fibo.dc                   //nome do arquivo em que se encontra a
                              // implementação da classe
```

Figura 5.8 Conteúdo do arquivo Fibo.apl da aplicação Fibonacci

O segundo arquivo é o arquivo que conterá a declaração e definição da classe distribuída fibo.dc. Nessa declaração consta a declaração do método construtor e destrutor da classe Fibo, além da declaração do método calculate que recebe como argumento um número inteiro e devolve como resposta um número inteiro. É interessante que esse arquivo tenha o mesmo nome da classe distribuída que é definida nele, para fins de legibilidade, seguida da extensão **.dc** O conteúdo do arquivo Fibo.dc pode ser visto na Figura 5.9

O método calculate da classe Fibo recebe como um argumento um número inteiro **n**, se esse número for menor do que dois, retorna o valor 1. Caso o argumento passado seja maior ou igual a dois, o valor de retorno é o resultado da soma das chamadas recursivas da função calculate, sendo que em uma das chamadas se tem como argumento **n-1**, e na outra, tem-se como argumento **n-2**.

```
dclass Fibo                  //classe distribuída Fibo
{
 public:                      //métodos públicos
   Fibo ( );                  //construtor do objeto da classe Fibo
   ~Fibo ( );                 //destrutor do objeto da classe Fibo
   int calculate (int);       //método calculate da classe Fibo
};
Fibo::Fibo(void) { };        //implementação do método construtor
Fibo::~Fibo(void) { };       //implementação do método destrutor
int Fibo::calculate (int n)  //implementação do método calculate
{
 if (n<2)
      return (1);
 else return (calculate(n-1) + calculate (n-2));
}
```

Figura 5.9 Conteúdo do arquivo Fibo.dc da aplicação Fibonacci

O último arquivo da aplicação é o arquivo principal da aplicação, chamado de Main.cc. Como esse arquivo define a aplicação principal, esse arquivo deve incluir todos os cabeçalhos das bibliotecas necessárias pelo programa, nesse caso foi incluído o cabeçalho da biblioteca **stdio.h** que é a biblioteca padrão do C++ para entrada e saída, além das classes distribuídas, nesse caso foi incluído o arquivo **Fibo.dc**, pois é nesse arquivo que está contida a classe distribuída Fibo. O Conteúdo detalhado desse arquivo pode ser visto na Figura 5.10

```
#include <stdio.h>               //inclui a biblioteca de E/S padrão do C
#include "fibo.dc"               //inclui a classe distribuída "fibo.dc"

int fibo2 (Fibo *f1, Fibo *f2, int n)       //função que retorna o cálculo
{
  return (f1->calculate (n-1) + f2->calculate (n-2));
}
void main ()                     //função principal do programa
{
  Fibo f1, f2;                   //instancia dois objetos da classe FIBO
  printf ("%d\n", fibo2(&f1, &f2, 5));       //imprime o resultado
}
```

Figura 5.10 Conteúdo do arquivo Main.cc da aplicação Fibonacci

Note que o paralelismo da aplicação se deve ao fato de serem instanciados dois objetos da classe distribuída Fibo (**f1** e **f2**). Na linha **printf ("%d\n", fibo2(&f1, &f2, 5));**, é chamada a função fibo2 que está implementada no programa principal, que distribui o trabalho do cálculo da série de Fibonacci. É importante observar que o símbolo **&** na frente dos objetos **f1** e **f2** significa que se está passando o endereço do objeto (ponteiro) para a função. Isso se deve ao fato de se ter instanciados dois objetos da classe Fibo. Uma variação para isso, seria a definição de dois ponteiros para objetos da classe Fibo e, ao invés de se passar como parâmetro o objeto antecedido do símbolo **&**, seriam passados como parâmetro o próprio ponteiro.

A função **fibo2** recebe como argumentos dois ponteiros para objetos da classe **Fibo**, (representados pelo símbolo *), além de um número inteiro e retorna como valor de resposta um número inteiro. Na linha que contém o trecho de código: **return (f1->calculate (n-1) + f2->calculate (n-2));** , o símbolo **->** é usado quando se quer referenciar um método de um objeto o qual se faz aceso via ponteiro.

Após a definição de todos os arquivos da aplicação, deve-se passar para o passo de compilação, bastando para isso, digitar de dentro do diretório onde se encontram os arquivos, **apl Fibo.apl**, seguido da tecla Enter. Para executar a aplicação resultante basta digitar **./a.out** .

5.4 Classificação em DPC++

O objetivo desse exemplo de programa é mostrar uma das possíveis modelagens de um programa distribuído a fim de efetuar uma classificação de números inteiros. Nesse exemplo será utilizado o algoritmo de classificação *Bubblesort*. Apesar desse algoritmo apresentar uma complexidade quadrática, ou seja $O(n^2)$, sabendo que os melhores algoritmos de classificação seqüencial são da ordem de $O(n\log_2 n)$, apenas se está objetivando um exemplo ilustrativo.

Nesse exemplo a entrada do programa é um arquivo de números desordenados, denominado de **teste**. Esse arquivo **teste**, deve conter números inteiros separados pelo símbolo de quebra de linha (basta pressionar a tecla "enter") e não deve conter nenhum símbolo adicional após a edição do último número, pois esse símbolo adicional será encarado pelo programa como um número e, provavelmente, ocorreria um erro durante a execução do programa de ordenação.

A saída do programa é um arquivo chamado **saida**. Esse arquivo possui como conteúdo os mesmos números que se desejava ordenar, só que em ordem cresecente de ordenação.

Nesse exemplo de aplicação, o programa principal possui um vetor para a armazenagem dos números que se deseja ordenar. A classificação desses números é efetuada da seguinte forma: são instanciados três objetos pertencentes à classe **DSORT**. Esses objetos trabalham em conjunto, um deles classifica os primeiros $n/2$ números, onde n é a quantidade total de números que se deseja ordenar, o segundo classifica a partir dos $n/2+1$ números até o fim do vetor e o terceiro faz a combinação das duas metades classificadas, obtendo-se como resultado um vetor completamente classificado.

Uma observação importante é que nesse exemplo são utilizados apenas 2 objetos para se realizar a classificação propriamente dita. Isso se deve apenas ao caráter ditático desse material, podendo o programador instanciar quantos objetos desejar. Cabe ressaltar que esse exemplo foi pensado exclusivamente para dois objetos, além de que para uma quantidade muito pequena de números, a solução seqüencial é mais eficiente do que a solução distribuída. Esse fato se deve ao tempo de criação dos objetos no ambiente distribuído, sendo seu uso justificado para quantias grandes de dados.

Para a implementação desse programa, preceisa-se editar 4 arquivos, sendo eles: integersort.apl, tipos.h, dsort.dc e integersort.cc.

Como já foi visto nos exemplos anteriores, pode-se usar qualquer programa de edição de arquivos (*vi, pico, emacs, mcedit*,..). Um exemplo da edição desses arquivos no editor emacs seria digitar na tela de prompt **emacs** seguido da tecla "enter". Os comandos do editor e seu modo de operação não serão tratados nesse trabalho. Sob determinado ponto de vista, normalmente os iniciandos no ambiente Linux preferem utilizar o *mcedit* ou *emacs*.

O primeiro arquivo editado é o arquivo integersort.apl, que é responsável pelas informações dos arquivos constituintes da aplicação e das máquinas do ambiente distribuído. O conteúdo do arquivo integersort.apl pode ser visualizado na Figura 5.11. Note que tudo o que vier após o símbolo **//** é comentário do código pertencente à linha (não deve ser incluído no arquivo, pois o compilador irá acusar um erro em tempo de compilação)

```
start: integersort.cc       //nome do arquivo principal da aplicação
in:    verissimo            //máquina onde irá rodar a aplicação principal
with   verissimo, quintana, dionelio //máquinas integrantes do ambiente
                                     //distribuído
dclass: DSORT               //nome da classe distribuída
at:     dsort.dc            //arquivo onde se encontra a classe distribuída
```

Figura 5.11 Conteúdo do arquivo integersort.apl da aplicação Integersort

O segundo arquivo editado é o arquivo **tipos.h**. Esse arquivo tem a finalidade de definir alguns tipos abstratos de dados que são usados pelo programa. A linha inicial desse arquivo, é para o caso de que existem dois arquivos (dsort.dc e integersort.cc) que possuêm o **tipos.h** em seu cabeçalho. Como esses arquivos incluem o mesmo arquivo (**tipos.h**), as linhas iniciais têm a finalidade de garantir que o compilador não redefina os tipos uma segunda vez, o que geraria um erro de compilação.

A estrutura que é definida nesse arquivo é a estrutura **tarray**, que é composta por uma variável **vetor[ ]** e uma variável **cont**. A variável **vetor[ ]** é um array de inteiros cujo comprimento é dependente da variável **MAXVETOR**, que nesse caso é 1000. A variável **cont** que é do tipo inteiro, possui a finalidade de armazenar a quantidade de elementos presentes na variável **vetor[ ]**.

Nesse arquivo também são definidas quatro funções: **meio**, **tamanho** , **valf1**, **vali2**. A função **int meio(int)** é uma função do tipo inteiro que recebe como parâmetro um inteiro correspondente ao tamanho de um vetor e retorna a posição central desse vetor. A função **int tamanho(int, int)** recebe como entrada um valor inicial e um final de um vetor, retornando um valor inteiro correspondente ao tamanho do vetor. As funções **int valf1(int, int)** e **int vali2(int, int)** são funções que desempenham a função de "dividir" um vetor em dois menores. A função **int valf1(int, int)** recebe como argumento o valor inicial e final de um vetor e retorna o valor final (valf1) de um hipotético vetor1. A função **int vali2(int, int)**

recebe como argumento o valor inicial e final de um vetor e retorna o valor inicial (vali2) de um hipotético vetor2. Note que essas funções não dividem o vetor em duas estruturas e sim devolvem ponteiros para emular dois vetores em uma única estrutura.

```
#ifndef TIPOS_H
#define TIPOS_H
#define MAXVETOR 1000
struct tarray
{
 int vetor[MAXVETOR];   //vetor de inteiros
 int cont ;              //inteiro
};
int meio(int tamanho)
{
  int m;
  m = (tamanho / 2) - 1;
  if ((tamanho % 2) != 0)
    m++;
  return m;
}
int tamanho(int i, int f)
{
  int tam;
  tam = (f - i) + 1;
  return tam;
}
int valf1(int i1,int f2)
{
  return(meio(tamanho(i1, f2)) + i1);
}
int vali2(int i1, int f2)
{
  return(valf1(i1,f2) + 1);
}
#define tarray_t struct tarray
#endif
```

Figura 5.12 Conteúdo do arquivo tipos.h da aplicação Integersort

O terceiro arquivo editado é o arquivo **dsort.dc**. É esse arquivo que contém a definição e implementação da classe distribuída **DSORT**. Esse arquivo inclui o cabeçalho do arquivo **tipos.h** a fim de que os tipos abstratos de dados definidos naquele arquivo possam ser usados. A classe **DSORT** possui um método construtor (**DSORT ();**) e um método destrutor (**~DSORT ();**). Também possui dois métodos. O primeiro método **tarray_t combina (tarray_t, tarray_t)** recebe como parâmetro de entrada duas estruturas do tipo **tarray** (definida no arquivo **tipos.h**) e devolve outra estrutura do tipo **tarray**. Essa função é necessária para combinar dois vetores previamente ordenados em um único vetor totalmente ordenado. O segundo método **tarray_t classifica (tarray_t vetor, int, int)** serve para classificar um vetor previamente desordenado. Esse método recebe como parâmetro de entrada uma estrutura do tipo **tarray** e devolve uma estrutura do tipo **tarray**, cujo vetor está ordenado.

```
#include "tipos.h"

class DSORT
{
```

```
 public:
   DSORT () {};
   ~DSORT() {};
   tarray_t combina  (tarray_t, tarray_t);
   tarray_t classifica (tarray_t vetor, int, int );
};
tarray_t DSORT::combina (tarray_t vetor1, tarray_t vetor2)
{
 tarray_t auxvetor;
 int i = 0;
 int j = 0;
 int k = 0;
 int y;
 auxvetor.cont =vetor1.cont+vetor2.cont;
 while ((i < vetor1.cont) && (j < vetor2.cont))
   {
     if (vetor1.vetor[i] < vetor2.vetor[j])
       {
         auxvetor.vetor[k++] = vetor1.vetor[i++];
       } else
           {
             auxvetor.vetor[k++] = vetor2.vetor[j++];
           }
   }
 if (i < vetor1.cont)
   {
     for (y = i; y < vetor1.cont; y++)
        auxvetor.vetor[k++] = vetor1.vetor[y];
     } else
       {
         for (y = j; j <vetor2.cont; y++)
          auxvetor.vetor[k++] = vetor2.vetor[y];
       }
 return auxvetor;
}
tarray_t DSORT::classifica (tarray_t vetor, int i, int f)
{
  bool troca = true;
  int aux, j, x, tam, m;
  tarray_t auxvetor;
  int k = 0;
  for (j = i; j <=f; j++)
     {
       auxvetor.vetor[k++] = vetor.vetor[j];
     }
    tam = tamanho(i,f)-1;
  m = tam;
  auxvetor.cont = tamanho(i,f);
  if (auxvetor.cont == 1)
    return auxvetor;
  while (troca == true)
    {
      troca = false;
      for (j = 0; j < m; j++)
```

```
          if (auxvetor.vetor[j] > auxvetor.vetor[j+1])
           {
             aux = auxvetor.vetor[j];
             auxvetor.vetor[j] = auxvetor.vetor[j+1];
             auxvetor.vetor[j+1] = aux;
             troca = true;
           }
        m -=1;
      }
    return auxvetor;
  }
```

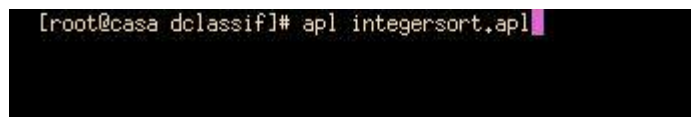
Figura 5.13 Conteúdo do arquivo dsort.dc da aplicação Integersort

O último arquivo é o arquivo principal da aplicação, chamado de Intergersort.cc. Esse arquivo inclui todos os cabeçalhos das bibliotecas necessárias pelo programa, nesse caso foi incluído o cabeçalho da biblioteca **stdio.h** que é a biblioteca padrão do C++ para entrada e saída, além da classe distribuída, nesse caso foi incluído o arquivo **dsort.dc**, pois é nesse arquivo que está contida a classe distribuída **DSORT**. O Conteúdo detalhado desse arquivo pode ser visto na Figura 5.10. No corpo do programa pode ser visto que são definidos dois ponteiros para arquivos chamados ***readFile** e ***outpFile** respectivamente. Também é definido uma variável **vetor** do tipo **tarray_t**, além de três instâncias da classe distribuída **DSORT**. Uma observação importante é que os comentários no código têm apenas um efeito ilustrativo, não devendo ser inseridos no código, pois o compilador DPC++ irá acusar erro.

O programa funciona em três etapas. A primeira etapa é onde é feita a inicialização do **vetor**. Essa inicialização se dá através do arquivo **teste**, sendo esse o arquivo que contém os números inicialmente desordenados. A segunda etapa é onde é realizada a classificação distribuída. São definidas duas novas estruturas **tarray_t** para que o resultado da classificação das partes do vetor, que é realizada por dois objetos distribuídos possa ser armazenada para uma futura combinação das mesmas. A terceira etapa tem a função de transferir os elementos ordenados do vetor para um arquivo denominado **"ord"**.

```
#include <stdio.h>
#include "dsort.dc"

int main ()
{
  FILE *readFile;
  FILE *outpFile;
  int j, i1, f2;
  tarray_t vetor;
  DSORT d1, d2, d3;

/* Inicio da Primeira Etapa */
  fprintf (stderr," Programa de ordenacao\n");
  readFile = fopen("teste", "rt");
  i1 = 0;
  f2 = 0;
  fscanf(readFile, "%d", &vetor.vetor[f2++]);
   while (!feof(readFile))
    {
      fscanf(readFile, "%d", &vetor.vetor[f2++]);
```

```
  }
 f2--;
 fclose(readFile);
/* fim da primeira etapa */

/* Inicio da Segunda etapa */
 int f1,i2;
 f1 = valf1(i1,f2);
 i2 = vali2(i1,f2);
 tarray_t vetor1,vetor2;

 vetor1 = d1.classifica(vetor, i1, f1);

 vetor2 = d2.classifica(vetor, i2, f2);

 vetor = d3.combina(vetor1 ,vetor2);
/* fim da Segunda etapa */

/*inicio da Terceira Etapa */
 outpFile = fopen("saida", "wt");
  j = 0;
 do
  {
   fprintf(outpFile,"%d\n",vetor.vetor[j++]);
   } while (j <= f2);
 fclose(outpFile);
/*Fim da Terceira Etapa */
}
```

Figura 5.14 Conteúdo do arquivo Integersort.cc da aplicação Integersort

Após a definição de todos os arquivos da aplicação, deve-se passar para o passo de compilação, bastando para isso, digitar de dentro do diretório onde se encontram os arquivos, **apl Integersort.apl**, seguido da tecla "enter", Como pode ser visto na Figura 5.15

Figura 5.15 Compilação da aplicação Integersort em DPC++

O resultado da compilação da aplicação pode ser visto na Figura 5.16. Note que o arquivo executável possui todos os atributo de exeução com o valor "x".

```
[root@casa dclassif]# l
total 18
drwxr-xr-x   2 root     root         1024 out 22 19:47 ./
drwxr-xr-x   3 root     root         1024 set  3 15:17 ../
-rwxr-xr-x   1 root     root         9863 out 22 19:44 a.out*
-rw-r--r--   1 root     root         1893 set  3 16:49 dsort.dc
-rw-r--r--   1 root     root          111 ago 11 16:29 integersort.apl
-rw-r--r--   1 root     root         1077 out 15 19:35 integersort.cc
-rw-r--r--   1 root     root          467 set  3 14:01 tipos.h
[root@casa dclassif]#
```

Figura 5.16 Executável da aplicação compilada

Ao invés de editar um arquivo manualmente, pode-se implementar um gerador de seqüencias de números desordenados. Nesse exemplo implementou-se o programa *gerafile*. Esse programa recebe como parâmetro de entrada um número (quantidade a ser gerada) e o nome de um arquivo (arquivo chamado **teste**). Como pode ser visto na Figura 5.17.

```
[root@casa dclassif]# ./gerafile 900 teste
```

Figura 5.17 Geração automática de um arquivo desordenado

A execução do programa *gerafile*, apresenta a seguinte tela, Figura 5.18. O resultado é um programa que servirá de entrada para a aplicação Integersort.

```
[root@casa dclassif]# ./gerafile 900 teste
Quantidade = 900
Nome do arquivo gerado = teste
[root@casa dclassif]#
```

Figura 5.18 Execução do gerador de seqüencia de números desordenados *gerafile*

O conteúdo do arquivo gerado deve ser uma seqüência de números, separados por uma quebra de linha (tecla "enter"). Um pedaço do arquivo gerado pelo programa *gerafile*, pode ser visto na Figura 5.19. Note que essa figura é a visualização do arquivo utilizando o editor *emacs*.

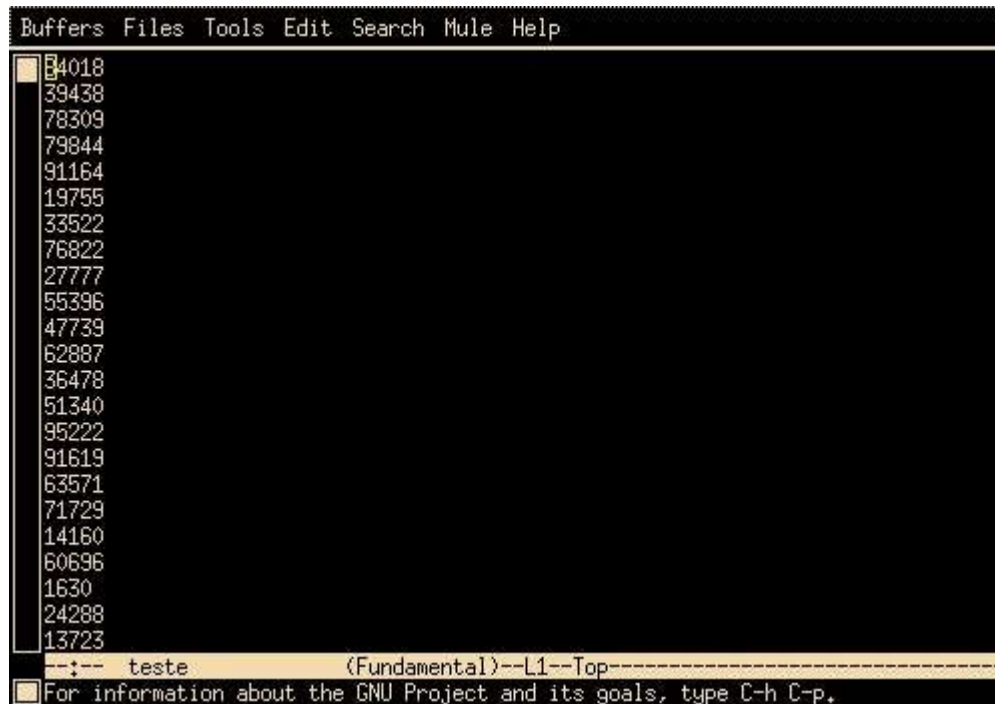



Figura 5.19 Visualização de um arquivo desordenado no editor *emacs*

Para executar a aplicação resultante basta digitar **./a.out**. O programa irá procurar, no diretório corrente do arquivo executável, por um arquivo chamado **teste** que é o nome do arquivo que se deseja ordenar. Caso o programa for executado sem o arquivo **teste** ocorrerá uma exceção de execução. O comando para executar a aplicação Integersort, pode ser visto na Figura 5.20.

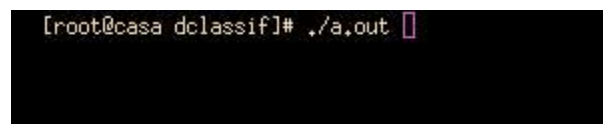
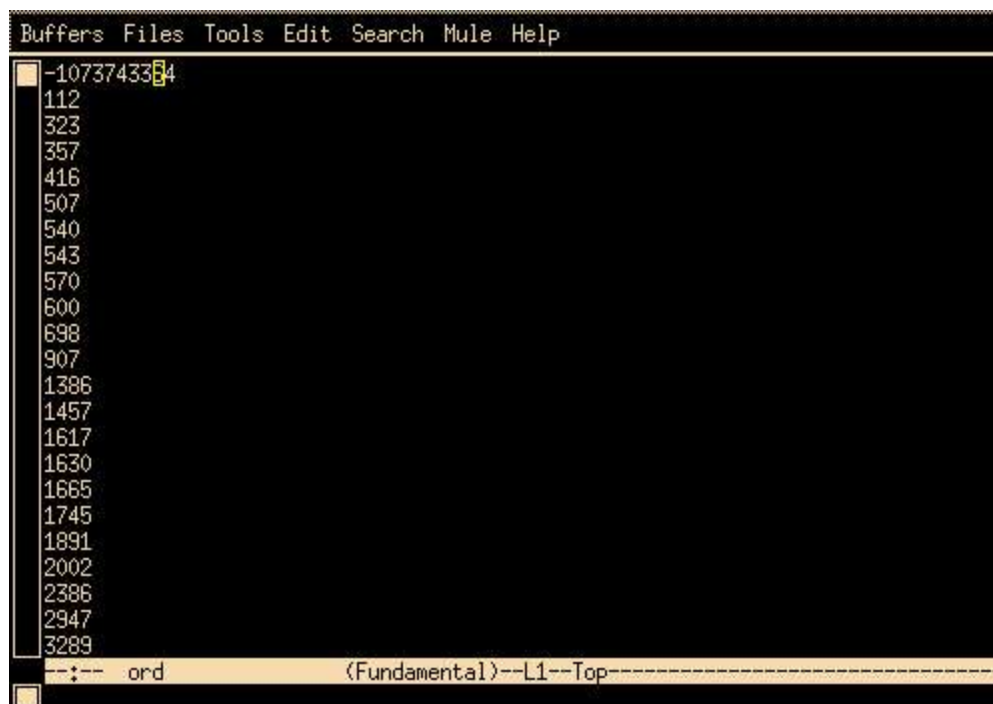



Figura 5.20 - Comando para a execução da aplicação Integersort

O conteúdo do arquivo ordenado são os números inicialmente desordenados no arquivo **teste**, só que agora ordenados em ordem crescente. Um pedaço do arquivo gerado pelo programa Integersort, pode ser visto na Figura 5.21. Note que essa figura é a visualização do arquivo utilizando o editor *emacs*.

Figura 5.21 Visualização do arquivo **saida** gerado pela aplicação Integersort

5.5 Conclusões

Nesse capítulo foram mostrados programas escritos na linguagem DPC++ a fim de exemplificar os conceitos discutidos no capítulo 4. Os exemplos apresentados foram validados e possiblitam, ao futuro iniciado, no ambiente que esse se sinta mais familiarizado com esse.

Algumas observações são muito importantes, entre elas é que a pessoa que vier a programar no ambiente DPC++ deve ter uma boa familiaridade com o paradigma de orientação a objetos e um bom conhecimento da linguagem C++. Alguns detalhes inerentes ao modelo distribuído adotado por DPC++, fazem com que muitas vezes, tenha-se que repensar nos objetos e métodos que vierem a ser implementados. Um exemplo típico disso é que o modelo adotado por DPC++ não permite a passagem de ponteiros e nem endereços de objetos e/ou estruturas de dados o que faz com que, muitas vezes, tenha-se que procurar alternativas para poder superar certas dificuldades.

6 Apêndice

Esse apêndice[0] tem a função de exemplificar a definição de aplicações tanto em PVM quanto em MPI.

6.1 Código em PVM

O código abaixo está escrito em C e faz uso das primitivas disponíveis pela biblioteca PVM, mostrando o emprego de tais diretivas.

```
1 #include <pvm3.h>
2 #include <stdio.h>
3
4 int tasks, child[5], cont, temp, h_status[3];
5 static char *hosts[ ] = {"scliar", "verissimo", "dionelio",};
6
7   void main (void)
8   {
9    pvm_addhosts(hosts, 3, h_status);
10  pvm_spawn("test", (char **) 0, PvmTaskDefault, (char*) 0, 5, child);
11    if (tasks < 5)
12      {printf("Erro ao inicializar as 5 tarefas\n"); return(0); }
13    for (cont = 0; cont < 5; cont++)
14      {
15         pvm_initsend (PvmDataDefault);
16         pvm_pkint (&child[cont], 1, 1);
17         Pvm_send (child[cont], 1);
18         pvm_recv(-1, -1);
19         pvm_upkint (&temp, 1, 1);
20         pvm_kill (child[cont]);
21     }
22   pvm_delhosts (hosts, 3, h_status);
23 }
```

Figura 6.1 Exemplo de código em PVM

Na Figura 6.1 tem-se um exemplo de programação em PVM. Na linha 5 tem-se o nome das máquinas que serão usadas no processamento. Umas das grandes características do PVM é que estas máquinas nem ao menos precisam ter a mesma arquitetura, ou seja, pode-se misturar máquinas SUN com Linux para executar uma tarefa. Na linha 9 tem-se um comando para inserir as referidas máquinas no ambiente e na linha 10, iniciam-se 5 tarefas de nome **"test"**, onde seus id's (numero para identificar o processo dentro da maquina virtual) são armazenados no vetor **child[ ]**. Os comandos do PVM retornam, geralmente um inteiro que indica condição de sucesso ou erro.

---

[0] Todo o conteúdo do apêndice foi retirado do trabalho **Núcleos de Comunicação: MPI e OpenMP,** escrito por Elgio Schlmer e está disponível em (www.inf.ufrgs.br/~elgio/trabs-html/Nucleo/openMP.html)

No caso do *pvm_spawn()*, este retorno é apenas o número de tarefas que efetivamente foram iniciadas. Se este número for menor do que o desejado (no caso, 5) deduz-se que houve um erro (linhas 11 e 12).

Nas linhas 14 a 22 temos simplesmente o envio do identificador de processo para cada processo, o recebimento de um inteiro deste processo e o comando para matá-lo. Para enviar uma mensagem com PVM, uma maratona de procedimentos são necessárias. A primeira delas é preparar o buffer de mensagem com o *pvm_initsend()*. A constante *PvmDataDefault* indica que o PVM deve formatar as mensagens por um padrão XDR. Este padrão possibilita enviar mensagens entre arquiteturas diferentes. Caso se utilize apenas um tipo de arquitetura (todos PC's rodando Linux, por exemplo), pode-se usar o parâmetro *PvmDataRow* que não provê nenhum tipo de codificação, sendo, portanto, muito mais eficiente.

Na linha 16, tem-se um comando de empacotamento de mensagens. Podemos, numa única mensagem, enviar vários tipos de dados, como inteiros, strings, etc. Após a mensagem estar devidamente preparada para ser enviada, o comando *pvm_send()* se encarrega de enviá-la ao processo destino.

Para receber uma mensagem, usa-se o *pvmrecv()*. No exemplo (linha 18) o primeiro **-1** indica o recebimento de mensagem de qualquer processo. Se deseja-se receber apenas de um único, coloca-se seu id neste parâmetro. O outro **-1** indica o recebimento de qualquer tag (veja na linha 17 o envio de msg com tag=**1** - segundo parâmetro). O conceito de tag pode ser usado para separar, por exemplo, mensagens de dados das de controle (tag=**1** para dados e tag=**2** para controle).

Novamente, na linha 19, deve-se utilizar rotinas do PVM para "desempacotar" os dados na mesma ordem com que foram empacotados.

Na linha 20 tem-se o comando para matar o processo para quem acabou de enviar a mensagem e, finalmente na linha 22 o comando para desmontar a máquina paralela virtual.

6.2 Código em MPI

O código abaixo está escrito em C e faz uso das primitivas disponíveis pela biblioteca MPI, mostrando o emprego de tais diretivas.

```
1 #include <mpi.h>
2 #include <stdio.h>
3
4 int mpierr, rank, size;
5 MPI_Status status;
6 char *msg = "OK";
7        temp[10];
8
9  void main (void)
10 {
11   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12   MPI_Comm_size(MPI_COMM_WORLD, &size);
13   if (rank == 0)
14     for (cont = 1; cont<size; cont++)
15      {
16        MPI_Send (msg, 2, MPI_CHAR, cont, 1, MPI_COMM_WORLD);
17        MPI_Recv (temp, 10, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &status);
18      }
19      else
20        {
21          MPI_Recv (temp, 10, MPI_CHAR, 0,1, MPI_COMM_WORLD, &status);
22          MPI_Send(msg, 2, MPI_CHAR, 0, 1, MPI_COMM_WORLD);
23          MPI_Finalize();
24        }
25
26   }
```

Figura 6.2 Exemplo de código em MPI

Neste exemplo, n cópias do processo são disparadas em *n* nodos. Inicialmente, na linha 11, o código registra-se no ambiente e obtém em **rank** seu identificador. Como todos os processos são criados na linha de comando, não existe uma hierarquia de processos (não existe um processo "pai"). Todos são numerados de "0" a "*n*-1", sendo *n* o número total de processos que deve ser menor ou igual ao número de nodos disponíveis. Na Figura 6.2, determina-se que o nodo de número 0 será responsável para enviar mensagens aos demais. O envio e recebimento de mensagens não é muito diferente do PVM, apenas nota-se a não necessidade de "empacotar-se" as mensagems mas, em contrapartida, deve-se informar (linha 16, parâmetro **MPI_CHAR**) qual é o tipo de dado que está se enviando.

7 Bibliografia

[AME85] AMERICA, P. Design Issue in a Parallel Object-Oriented Language. In: M. Feilmeier; G. Joubert and U. Schendel, editors, International Conference in Parallel Computing, p. 325-330. North-Holland. 1985 .

[AMZ95] AMZA, Cristiana et. al. TreadMarks: Shared Memory Computing on Networks of Workstations. ACM Computer. 1995.

[ÁVI99a] ÁVILA, Rafael B., NAVAUX, Philipe O. A. Um Modelo de Expressão Implícita de Concorrência Aplicado à Programação Orientada a Objetos. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 3., 1999, Porto Alegre. **Anais...** Porto Alegre: UFRGS, 1999. P.225-229.

[ÁVI99b] ÁVILA, Rafael B., NAVAUX, Philipe O. A. A Fine-Grain Concurrency Model for Object-Oriented Distributed Applications. Artigo submetido ao EUROPAR'99, Toulouse, França, 1999.

[ÁVI99c] ÁVILA, Rafael B.; NAVAUX, Philippe O. A. Um modelo de paralelismo de grão fino para objetos distribuídos. Porto Alegre: PPGC, UFRGS. 1999. Dissertação de Mestrado.

[BAL89] BAL, H. E.; STEINER, J. G.; TANENBAUM, A. S. Programming Languages for Distributed Computing Systems. **Computing Surveys.** ACM, v.21, n.3, p.261-320, New York: ACM, Sept.1989.

[BOD95] BODEN, N.et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, Los Alamitos, v.15, n.1, p.29-36, Feb. 1995

[BAR98] BARRETO, Marcos E., NAVAUX, Philipe O. A.; RIVIÈRE, Michel. P. DECK: a New Model for a Distributed Executive Kernel Integrating Communication and Multithreading for Support of Distributed Object Oriented Application with Fault Tolerance Support. In: CONGRESSO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 4., 1998, Neuquén, Argentina. **Anales...** Neuquén: Universidad Nacional de Comahue, Facultad de Economia y Administración, Departamento de Informática y Estadística, 1998. v.2, p.623-637.

[CAV92] CAVALHEIRO, G. G. **Implementação de Concorrência em C++** Porto Alegre: CPGCC da UFRGS, 1992. (Trabalho Individual) .

[CAV93] CAVALHEIRO, G. G. H.; NAVAUX, P.O.A. Um Modelo Distribuído para Linguagens Orientadas a Objetos. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 20., 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993. v.2, p.518-532.

[CAV93a] CAVALHEIRO, G. G. H.; NAVAUX, P. O. A. DPC++: uma linguagem para processamento distribuído. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES -- PROCESSAMENTO DE ALTO DESEMPENHO, 5., 1993, Florianópolis. Anais... Florianópolis: SBC, 1993. v.2, p.732-744.

[CAV94] CAVALHEIRO, Gerson G. H; Um modelo para linguagens orientadas a objetos distribuídos. Porto Alegre: CPGCC da UFRGS, 1994. Dissertação de Mestrado.

[CAV95] CAVALHEIRO, G. G. H.; KRUG, R. C.; RIGO, S. J.; NAVAUX, P. O. A. DPC++: an object-oriented distributed language. In: CONFERENCIA INTERNACIONAL DE LA SOCIEDAD CHILENA DE CIENCIA DE LA COMPUTACIÓN, 15., 1995,

Arica, CL. **Actas...** Santiago: Sociedad Chilena de Ciencia de la Computación, 1995. p.92-103.

[CHA97] CHAIMOWICZ, Luiz; ÁRABE, José Nagib Cotrim. Balanceamento de carga em um ambiente Distribuído. In IX Simpósio Brasileiro de Arquitetura de Computadores e Processamento de alto desempenho. 1997. ANAIS. p. 475-491.

[DER97] De ROSE, C. A. F.; NAVAUX, P. Um Modelo Distribuído para a Alocação e Gerência de Processadores em Multicomputadores. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES-PROCESSAMENTO DE ALTO DESEMPENHO, 9., 1997, Campos do Jordão. **Anais...** São Paulo: Escola Politécnica da USP, 1997. p.447-461.

[DER99] De ROSE, C.A. F. ; MAI, C. G. Arquiteturas Paralelas Versáteis e de Baixo Custo para a Pesquisa e o Ensino na Área de Processamento Paralelo e Distribuído. Porto Alegre: PUCRS. 1999

[DIL95] DILLON, E.; SANTOS, C. Gamboa dos; GUYARD, J. Homogeneous and heterogeneous networks of workstations: message passing overhead. In: MPI DEVELOPERS CONFERENCE ´95, 1995, Motre-Dame, **Proceedings...** 1995.

[DUM95] DUMAS, Arthur. Programming WinSock. Sams Publishing, 1995.

[ELL90] ELLIS, M.; STROUSTRUP, B. **"The Annotated C++ Reference Manual."** Murray Hill: Addison-Weskey, 1990. 447p.

[EIC95] EICKEN, T. von; BASU, A.; BUCH V. Low-Latency Communication Over ATM Networks Using Active Messages. IEEE Micro. p.46-53. Feb. 1995.

[FLY92] FLYNN, M. J. Some Computer Organizations and their Effectiveness. **IEEE Transactions on Computers,** New York, v. C-21, n.9, p.948-160, Sept. 1972.

[FRO98] FRÖHLICH, Antonio Augusto Medeiros. **SNOW project**. Disponível por WWW em http://www.first.gmd.de/~guto/snow (fev. 1998)

[GEI94] GEIST, Al et al. **PVM: parallel virtual machine**. Cambridge, MA: MIT Press, 1994.

[GEI94a] GEIST, A.; BAGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing. MIT Press. 1994.

[GRO96] GROPP, W. et al. A high-performance, portable implementation of the MPI message passing interface standard. **Parallel Computing**, v.22, n.6, p.789-828, Sep. 1996.

[HEI98] HEISS, Hans-Ulrich. **AG Heiβ-project Arminius**. Disponível por WWW em http:/www.unipaderborn.de/cs/heiss/arminius (dez. 1998).

[HIP97] HIPPER, G.; TAVANGARIAN, D. Advanced workstation cluster architectures for parallel computing. **Journal of Systems Architecture**, Amsterdam, v.44, n.3/4, p.207-226, Dec. 1997.

[HUF89] HUFNAGEL, S. P; BROWNE, J. C. Desempenho Properties of Vertically Partioned Object-Oriented Systems. **IEEE Transaction on Software Enginnering**, New York: v.32, n.4, :p.935-946, Aug. 1989.

[HWA93] HWANG K. Advanced computer architecture: parallelism, scalability, programmability. MacGraw-Hill Series in Computer Science. McGraw-Hill. 1993.

[IEE95] INSTITUTE OF ELECTRICAL AND ELETRONIC ENGINEERS. **Local and metropolitan area networks-supplement--media acess control (MAC)**

parameters, physical layer, medium attachment units and repeater for 100Mb/s operation, type 100BASE-T (clauses 21-30), IEEE 802.3u-1995. New York, NY, 1995.

[JAL94] JALOTE, P. Fault tolerance in distributed systems. Englewood Cliffs: PTR Prentice Hall, 1994. 432p.

[KRU94] KRUEGER, Phillip; SHIVARATRI, Niranjan G. Adaptative Location Policies for global Scheduling. **IEEE Transactions on Software Engeneering.** Vol 20, No 6. Junho 1994. Pp 432-444.

[LAU97] LAURIA, Mario; CHIEN, Andrew. MPI-FM: high performance MPI on workstation clusters. **Journal of Parallel and Distributed Computing**, Orlando, FL, v.40, n.1, p.4-18, Jan. 1997.

[MEY88] MEYER, B. **Objected-Oriented Software Construction**. New York 1988. 534p.

[MPI94] MPI FORUM. **The MPI message passing interface standard.** Knoxville: University of Tenessee, 1994.

[MPI97] MESSAGE PASSING INTERFACE FORUM. **MPI-2: Extensions to the Message-Passing Interface.** 1997.

[NAV97] NAVAUX, P. O. A. Manual de Instalação do Ambiente de Compilação DPC++. Porto Alegre: CPGCC da UFRGS, 1997.

[OHI96] OHIO SUPERCOMPUTER CENTER. **MPI primer/developing with LAM.** [S.l.]: Ohio State University, 1996.

[PIL97] PILLA, M. L.; BARRETO, M. E.; SANTOS, R. R. ; CAVALHEIRO, G. G. H.; NAVAUX, P. O. A. Mecanismo de tolerância a falhas para a linguagem distribuída DPC++. In: SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES -- PROCESSAMENTO DE ALTO DESEMPENHO, 9., 1997, Campos do Jordão. **Anais...** São Paulo: Escola Politécnica da USP, 1997. p.139-152.

[PIL97a] PILLA, M. L.; BARRETO, M. E.; SANTOS, R. R.; CAVALHEIRO, G. G. H.; NAVAUX, P. O. A. Implementação de um algortimo de criação de checkpoints para a linguagem distribuída DPC++. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 3., 1997, La Plata, AR. **Anales...** La Plata: Universidad Nacional de La Plata, 1997.

[PRY98] PRYLLY, L.; TOURANCHEAU, B. BIP: a new protocol designed for high performance networking on myrinet. In: IPPS/SPDP'98 WORKSHOPS, 10., 1998. **Proceedings...** Springer, 1998. P.472-485. (Lecture Notes in Computer Science, v.1388).

[SAN93] SANTOS, Rafael Ramos dos; CAVALHEIRO, Gerson; NAVAUX, Philippe O. A. "Mecanismo de Transporte para Comunicao entre Objetos Distribudos" IN: TELEMATICA 93, Simpsio de Redes de Computadores e suas Aplicacoes, SUCESU- RS, Porto Alegre, RS, 3 a 5 de agosto, 1993, Comunicao.

[SAN93a] SANTOS, R. R.; NAVAUX P. O. A. Um modelo Distribuído para Programação Orientada a Objetos. In: SALÃO DE INICIAÇÃO CIENTÍFICA, 5., 1993, Porto Alegre. **Resumos...** Porto Alegre: PROPESP/UFRGS, 1993. v.1, p.86.

[SAN96] SANTOS, R. R.; CAVALHEIRO, G. G. H.; NAVAUX, P. O. A. Checkpoints distribuídos em DPC++. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 2., 1996, San Luis, AR. **[Trabajos Seleccionados]...** San Luis: Universidad Nacional de San Luis, 1996. p.74-85.

[SNY86] SNYDER, A. Encapsulation and Inheritance in Object-Oriented Programming Languages. **SIGPlan Notices**, v.21, n.11, p.38-45, Nov. New York: ACM, 1986.

[SNY93] SNYDER, A. The Essence of Objects: Concepts and Terms. **IEEE Software**, v.10, n.1, p.31-42, Jan. New York: IEEE, 1993. .

[TAK 88] TAKEHASHI, T. Introdução a Programação Orientada a Objetos, In: EBAI, 3, Curitiba, jan 1988. .

[TAK9O] TAKAHASHI, T.; LIESENBERG, K. E. **Programação Orientada a Objet**os. São Paulo: IME-USP, 1990. 340p.

[WAR98] WARSCHKO, Thomas M.; BLUM, Joachin M.; TICHY, Walter F. PULC: ParaStation user-level comunication: design and overview. In: IPPS/SPDP'98 WORKSHOPS, 10., 1998. **Proceedings...** Springer, 1998. P.498-509. (Lecture Notes in Computer Scinece, v. 1388).

[WIL93] WILLEBEEK-LEMAIR, Marc H; REEVES, Anthony P. Strategies for Dynamic Load Balancing on Highly Parallel Computers. **IEEE Transactions on Parallel and Distributed Systems.** Vol 4, No 9. Set 93. Pp 979-992.

[WYA92] WYATT, B. B.; KAVI, K; HUFNAGEL, S. The Essence of Objects: Concepts and Terms. **IEEE Software**, v.9, n.6, p.56-66, Nov. New York:IEEE , 1992.

[YAU92] YAU S. S.; JIA, X.; BAE, D.-H. Software Design Methods for Distributed Computing Systems.**Computer Communications**, v.15, n.4, p.213-224, May London: 1992.